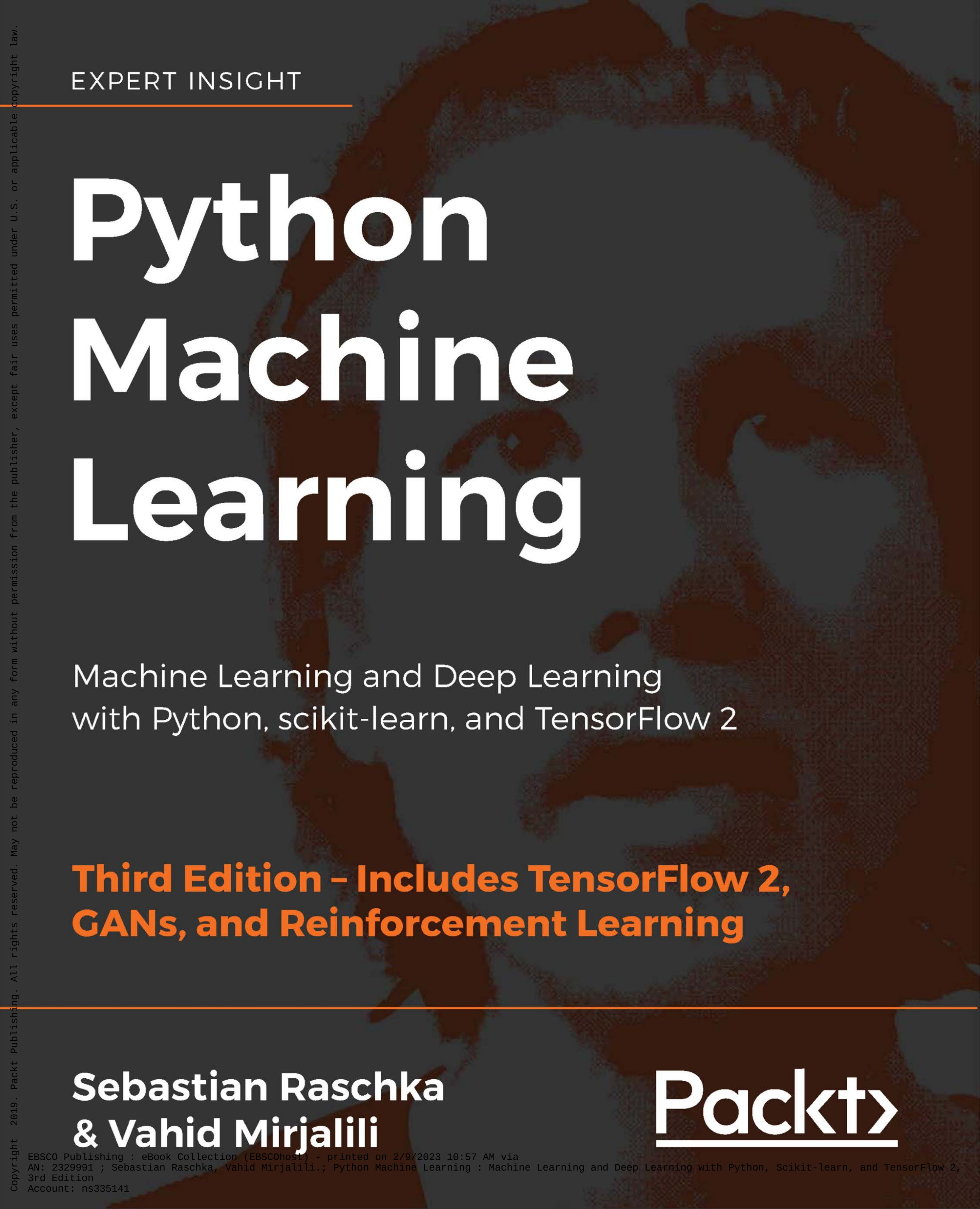


EXPERT INSIGHT

Python Machine Learning

Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2

Third Edition – Includes TensorFlow 2, GANs, and Reinforcement Learning

Sebastian Raschka & Vahid Mirjalili

Packt

Python Machine Learning

Third Edition

Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2

Sebastian Raschka

Vahid Mirjalili

Packt>

BIRMINGHAM - MUMBAI

Python Machine Learning
Third Edition

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Jonathan Malysiak
Acquisition Editor – Peer Reviews: Suresh Jain
Content Development Editors: Joanne Lovell, Chris Nelson
Technical Editor: Saby D'silva
Project Editor: Radhika Atitkar
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Presentation Designer: Sandip Tadge

First published: September 2015
Second edition: September 2017
Third edition: December 2019

Production reference: 1091219

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78995-575-0

www.packt.com

Contributors

About the authors

Sebastian Raschka received his doctorate from Michigan State University, where he focused on developing methods at the intersection of computational biology and machine learning. In the summer of 2018, he joined the University of Wisconsin-Madison as Assistant Professor of Statistics. His research activities include the development of new deep learning architectures to solve problems in the field of biometrics.

Sebastian has many years of experience with coding in Python and has given several seminars on the practical applications of data science, machine learning, and deep learning over the years, including a machine learning tutorial at SciPy, the leading conference for scientific computing in Python.

Among Sebastian's achievements is his book *Python Machine Learning*, which is a bestselling title at Packt and on Amazon.com. The book received the ACM Best of Computing award in 2016 and was translated into many different languages, including German, Korean, Chinese, Japanese, Russian, Polish, and Italian.

In his free time, Sebastian loves to contribute to open source projects, and methods that he implemented are now successfully used in machine learning competitions such as Kaggle.

> I would like to take this opportunity to thank the great Python community and the developers of open source packages who helped me create the perfect environment for scientific research and data science. Also, I want to thank my parents, who always encouraged and supported me in pursuing the path and career that I was so passionate about.
>
> Special thanks to the core developers of scikit-learn and TensorFlow. As a contributor and user, I had the pleasure of working with great people who are not only very knowledgeable when it comes to machine learning and deep learning but are also excellent programmers.

Vahid Mirjalili obtained his PhD in mechanical engineering working on novel methods for large-scale, computational simulations of molecular structures at Michigan State University. Being passionate about the field of machine learning, he joined the iPRoBe lab at Michigan State University, where he worked on applying machine learning in the computer vision and biometrics domains. After several productive years at the iPRoBe lab and many years in academia, Vahid recently joined **3M** Company as a research scientist, where he can use his expertise and apply state-of-the-art machine learning and deep learning techniques to solve real-world problems in various applications to make life better.

I would like to thank my wife, Taban Eslami, who has been very supportive and encouraged me on my career path. Also, special thanks to my advisors, Nikolai Priezjev, Michael Feig, and Arun Ross, for supporting me during my PhD studies, as well as my professors, Vishnu Boddeti, Leslie Kuhn, and Xiaoming Liu, who have taught me so much and encouraged me to pursue my passion.

About the reviewers

Raghav Bali is a senior data scientist at one of the world's largest healthcare organizations. His work involves the research and development of enterprise-level solutions based on machine learning, deep learning, and natural language processing for healthcare- and insurance-related use cases. In his previous role at Intel, he was involved in enabling proactive data-driven IT initiatives using natural language processing, deep learning, and traditional statistical methods. He has also worked in the finance domain with American Express, solving digital engagement and customer retention use cases.

Raghav has also authored multiple books with leading publishers, the most recent being on the latest advancements in transfer learning research.

Raghav has a master's degree (gold medalist) in information technology from the International Institute of Information Technology, Bangalore. Raghav loves reading and is a shutterbug, capturing moments when he isn't busy solving problems.

Motaz Saad holds a PhD in computer science from the University of Lorraine. He loves data and he likes to play with it. He has over 10 years of professional experience in natural language processing, computational linguistics, data science, and machine learning. He currently works as an assistant professor at the faculty of Information Technology, IUG.

Table of Contents

Preface

Through exposure to the news and social media, you are probably very familiar with the fact that machine learning has become one of the most exciting technologies of our time. Large companies, such as Google, Facebook, Apple, Amazon, and IBM, heavily invest in machine learning research and applications for good reason. While it may seem that machine learning has become the buzzword of our age, it is certainly not just hype. This exciting field opens up the way to new possibilities and has become indispensable to our daily lives. Think about talking to the voice assistant on our smartphones, recommending the right product for our customers, preventing credit card fraud, filtering out spam from our email inboxes, and detecting and diagnosing medical diseases; the list goes on and on.

Get started with machine learning

If you want to become a machine learning practitioner or a better problem solver, or maybe you are even considering a career in machine learning research, then this book is for you! For a novice, the theoretical concepts behind machine learning can be quite overwhelming, but the many practical books that have been published in recent years will help you to get started in machine learning by implementing powerful learning algorithms.

Practice and theory

Being exposed to practical code examples and working through example applications of machine learning are great ways to dive into this field. Also, concrete examples help to illustrate the broader concepts by putting the learned material directly into action. However, remember that with great power comes great responsibility!

In addition to offering hands-on experience with machine learning using the Python programming language and Python-based machine learning libraries, this book introduces the mathematical concepts behind machine learning algorithms, which are essential for using machine learning successfully. Thus, this book is different from a purely practical book; this is a book that discusses the necessary details regarding machine learning concepts and offers intuitive, yet informative, explanations on how machine learning algorithms work, how to use them, and, most importantly, how to avoid the most common pitfalls.

Why Python?

Before we dive deeper into the machine learning field, let's answer your most important question: "Why Python?" The answer is simple: it is powerful, yet very accessible. Python has become the most popular programming language for data science because it allows us to forget the tedious parts of programming and offers us an environment where we can quickly jot down our ideas and put concepts directly into action.

Explore the machine learning field

If you type "machine learning" as a search term into Google Scholar, it will return an overwhelmingly large number—3,250,000 publications. Of course, we cannot discuss all the nitty-gritty details of all the different algorithms and applications that have emerged in the last 60 years. However, in this book, we will embark on an exciting journey, covering all the essential topics and concepts to give you a head start in this field. If you find that your thirst for knowledge is not satisfied, you can use the many useful resources that this book references to follow up on the essential breakthroughs in this field.

We, the authors, can truly say that the study of machine learning made us better scientists, thinkers, and problem solvers. In this book, we want to share this knowledge with you. Knowledge is gained by learning, the key to this is enthusiasm, and the real mastery of skills can only be achieved through practice.

The road ahead may be bumpy on occasions, and some topics may be more challenging than others, but we hope that you will embrace this opportunity and focus on the reward. Remember that we are on this journey together, and throughout this book, we will add many powerful techniques to your arsenal that will help you to solve even the toughest problems the data-driven way.

Who this book is for

If you have already studied machine learning theory in detail, this book will show you how to put your knowledge into practice. If you have used machine learning techniques before and want to gain more insight into how machine learning actually works, this book is also for you.

Don't worry if you are completely new to the machine learning field; you have even more reason to be excited! This is a promise that machine learning will change the way you think about the problems you want to solve and show you how to tackle them by unlocking the power of data. If you want to find out how to use Python to start answering critical questions about your data, pick up *Python Machine Learning*. Whether you want to start from scratch or extend your data science knowledge, this is an essential and unmissable resource.

What this book covers

Chapter 1, *Giving Computers the Ability to Learn from Data*, introduces the main subareas of machine learning used to tackle various problem tasks. In addition, it discusses the essential steps for creating a typical machine learning model-building pipeline that will guide us through the following chapters.

Chapter 2, *Training Simple Machine Learning Algorithms for Classification*, goes back to the origin of machine learning and introduces binary perceptron classifiers and adaptive linear neurons. This chapter is a gentle introduction to the fundamentals of pattern classification and focuses on the interplay of optimization algorithms and machine learning.

Chapter 3, *A Tour of Machine Learning Classifiers Using scikit-learn*, describes the essential machine learning algorithms for classification and provides practical examples using one of the most popular and comprehensive open source machine learning libraries, scikit-learn.

Chapter 4, *Building Good Training Datasets – Data Preprocessing*, discusses how to deal with the most common problems in unprocessed datasets, such as missing data. It also discusses several approaches to identify the most informative features in datasets and how to prepare variables of different types as proper inputs for machine learning algorithms.

Chapter 5, *Compressing Data via Dimensionality Reduction*, describes the essential techniques to reduce the number of features in a dataset to smaller sets, while retaining most of their useful and discriminatory information. It also discusses the standard approach to dimensionality reduction via principal component analysis and compares it to supervised and nonlinear transformation techniques.

Chapter 6, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, discusses the dos and don'ts for estimating the performance of predictive models. Moreover, it discusses different metrics for measuring the performance of our models and techniques for fine-tuning machine learning algorithms.

Chapter 7, *Combining Different Models for Ensemble Learning*, introduces the different concepts of combining multiple learning algorithms effectively. It explores how to build ensembles of experts to overcome the weaknesses of individual learners, resulting in more accurate and reliable predictions.

Chapter 8, *Applying Machine Learning to Sentiment Analysis*, discusses the essential steps for transforming textual data into meaningful representations for machine learning algorithms to predict the opinions of people based on their writing.

Chapter 9, *Embedding a Machine Learning Model into a Web Application*, continues with the predictive model from the previous chapter and walks through the essential steps of developing web applications with embedded machine learning models.

Chapter 10, *Predicting Continuous Target Variables with Regression Analysis*, discusses the essential techniques for modeling linear relationships between target and response variables to make predictions on a continuous scale. After introducing different linear models, it also talks about polynomial regression and tree-based approaches.

Chapter 11, *Working with Unlabeled Data – Clustering Analysis*, shifts the focus to a different subarea of machine learning, unsupervised learning. It covers algorithms from three fundamental families of clustering algorithms that find groups of objects that share a certain degree of similarity.

Chapter 12, *Implementing a Multilayer Artificial Neural Network from Scratch*, extends the concept of gradient-based optimization, which we first introduced in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. In this chapter, we will build powerful, multilayer **neural networks** (**NNs**) based on the popular backpropagation algorithm in Python.

Chapter 13, *Parallelizing Neural Network Training with TensorFlow*, builds upon the knowledge from the previous chapter to provide a practical guide for training NNs more efficiently. The focus of this chapter is on TensorFlow 2.0, an open source Python library that allows us to utilize multiple cores of modern graphics processing units (GPUs) and construct deep NNs from common building blocks via the user-friendly Keras API.

Chapter 14, *Going Deeper – The Mechanics of TensorFlow*, picks up where the previous chapter left off and introduces the more advanced concepts and functionality of TensorFlow 2.0. TensorFlow is an extraordinarily vast and sophisticated library, and this chapter walks through concepts such as compiling code into a static graph for faster execution and defining trainable model parameters. In addition, this chapter provides additional hands-on experience of training deep neural networks using TensorFlow's Keras API, as well as TensorFlow's pre-made Estimators.

Chapter 15, *Classifying Images with Deep Convolutional Neural Networks*, introduces **convolutional neural networks** (**CNNs**). A CNN represents a particular type of deep NN architecture that is particularly well suited for image datasets. Due to their superior performance compared to traditional approaches, CNNs are now widely used in computer vision to achieve state-of-the-art results for various image recognition tasks. Throughout this chapter, you will learn how convolutional layers can be used as powerful feature extractors for image classification.

Chapter 16, *Modeling Sequential Data Using Recurrent Neural Networks*, introduces another popular NN architecture for deep learning that is especially well suited to working with text and other types of sequential data and time series data. As a warm-up exercise, this chapter introduces recurrent NNs for predicting the sentiment of movie reviews. Then, the chapter covers teaching recurrent networks to digest information from books in order to generate entirely new text.

Chapter 17, *Generative Adversarial Networks for Synthesizing New Data*, introduces a popular adversarial training regime for NNs that can be used to generate new, realistic-looking images. The chapter starts with a brief introduction to autoencoders, a particular type of NN architecture that can be used for data compression. The chapter then shows how to combine the decoder part of an autoencoder with a second NN that can distinguish between real and synthesized images. By letting two NNs compete with each other in an adversarial training approach, you will implement a generative adversarial network that generates new handwritten digits. Lastly, after introducing the basic concepts of generative adversarial networks, the chapter introduces improvements that can stabilize the adversarial training, such as using the Wasserstein distance metric.

Chapter 18, *Reinforcement Learning for Decision Making in Complex Environments,* covers a subcategory of machine learning that is commonly used for training robots and other autonomous systems. This chapter starts by introducing the basics of **reinforcement learning** (**RL**) to make you familiar with agent/environment interactions, the reward process of RL systems, and the concept of learning from experience. The chapter covers the two main categories of RL, model-based and model-free RL. After learning about basic algorithmic approaches, such as Monte Carlo- and temporal distance-based learning, you will implement and train an agent that can navigate a grid world environment using the Q-learning algorithm.

Finally, this chapter introduces the deep Q-learning algorithm, which is a variant of Q-learning that uses deep NNs.

What you need for this book

The execution of the code examples provided in this book requires an installation of Python 3.7.0 or newer on macOS, Linux, or Microsoft Windows. We will make frequent use of Python's essential libraries for scientific computing throughout this book, including SciPy, NumPy, scikit-learn, Matplotlib, and pandas.

The first chapter will provide you with instructions and useful tips to set up your Python environment and these core libraries. We will add additional libraries to our repertoire, and installation instructions are provided in the respective chapters, for example, the NLTK library for natural language processing in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, the Flask web framework in *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, and TensorFlow for efficient NN training on GPUs in *Chapter 13* to *Chapter 18*.

To get the most out of this book

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Download the example code files

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows

- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

Alternatively, if you have obtained a copy of the book from elsewhere or do not wish to create an account at Packt, all code examples are also available for download through GitHub at `https://github.com/rasbt/python-machine-learning-book-3rd-edition`.

All code in this book is also available in the form of Jupyter notebooks, and a short introduction can be found in the code directory of *Chapter 1, Giving Computers the Ability to Learn from Data*, at `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch01#pythonjupyter-notebook`. For more information about the general Jupyter Notebook GUI, please see the official documentation at `https://jupyter-notebook.readthedocs.io/en/stable/`.

While we recommend using Jupyter Notebook for executing code interactively, all code examples are available in both a Python script (for example, `ch02/ch02.py`) and a Jupyter Notebook format (for example, `ch02/ch02.ipynb`). Furthermore, we recommend that you view the `README.md` file that accompanies each individual chapter for additional information and updates (for example, `https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch01/README.md`).

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you to better understand the changes in the output. You can download this file from `https://static.packt-cdn.com/downloads/9781789955750_ColorImages.pdf`. In addition, lower resolution color images are embedded in the code notebooks of this book that come bundled with the example code files.

Conventions used

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown as follows: "And already installed packages can be updated via the `--upgrade` flag."

A block of code is set as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='x', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='o', label='versicolor')
>>> plt.xlabel('sepal length')
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Any command-line input or output is written as follows:

```
> dot -Tpng tree.dot -o tree.png
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `www.packtpub.com/support/errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the **Errata** section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1

Giving Computers the Ability to Learn from Data

In my opinion, **machine learning**, the application and science of algorithms that make sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using self-learning algorithms from the field of machine learning, we can turn this data into knowledge. Thanks to the many powerful open source libraries that have been developed in recent years, there has probably never been a better time to break into the machine learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.

In this chapter, you will learn about the main concepts and different types of machine learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using machine learning techniques for practical problem solving.

In this chapter, we will cover the following topics:

- The general concepts of machine learning
- The three types of learning and basic terminology
- The building blocks for successfully designing machine learning systems
- Installing and setting up Python for data analysis and machine learning

Building intelligent machines to transform data into knowledge

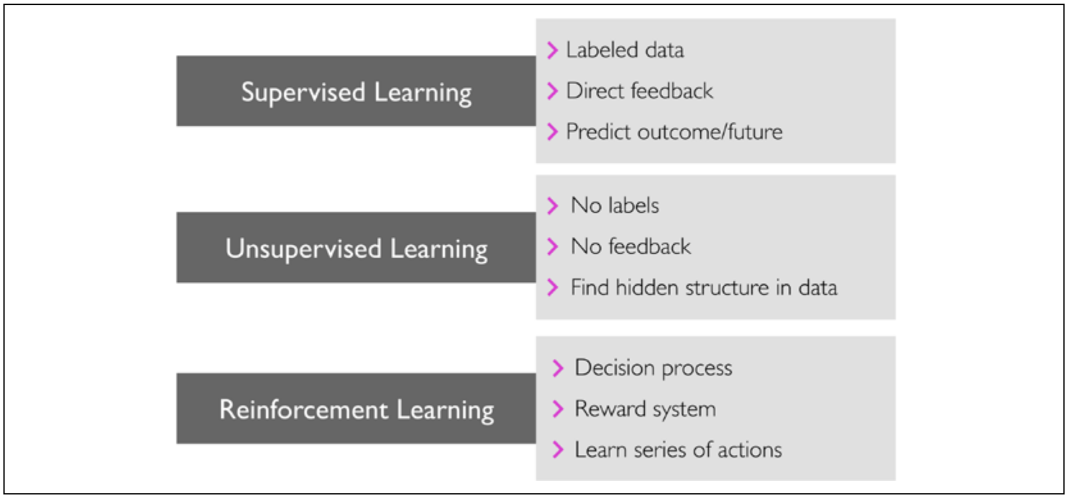
In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the 20th century, machine learning evolved as a subfield of **artificial intelligence** (**AI**) involving self-learning algorithms that derive knowledge from data in order to make predictions.

Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, machine learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models and make data-driven decisions.

Not only is machine learning becoming increasingly important in computer science research, but it is also playing an ever-greater role in our everyday lives. Thanks to machine learning, we enjoy robust email spam filters, convenient text and voice recognition software, reliable web search engines, and challenging chess-playing programs. Hopefully soon, we will add safe and efficient self-driving cars to this list. Also, notable progress has been made in medical applications; for example, researchers demonstrated that deep learning models can detect skin cancer with near-human accuracy (`https://www.nature.com/articles/nature21056`). Another milestone was recently achieved by researchers at DeepMind, who used deep learning to predict 3D protein structures, outperforming physics-based approaches for the first time (`https://deepmind.com/blog/alphafold/`).

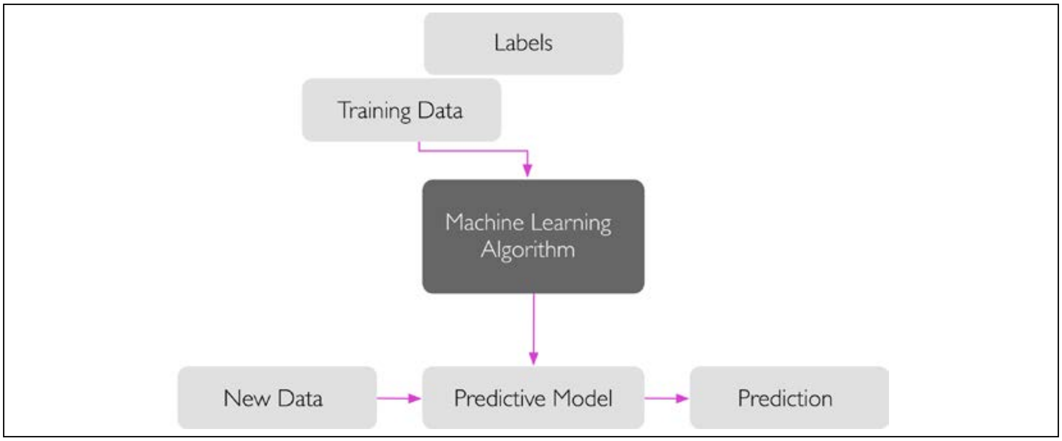
The three different types of machine learning

In this section, we will take a look at the three types of machine learning: **supervised learning**, **unsupervised learning**, and **reinforcement learning**. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an understanding of the practical problem domains where they can be applied:



Making predictions about the future with supervised learning

The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data. Here, the term "supervised" refers to a set of training examples (data inputs) where the desired output signals (labels) are already known. The following figure summarizes a typical supervised learning workflow, where the labeled training data is passed to a machine learning algorithm for fitting a predictive model that can make predictions on new, unlabeled data inputs:

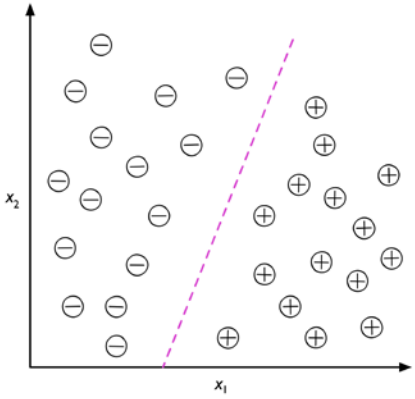


Considering the example of email spam filtering, we can train a model using a supervised machine learning algorithm on a corpus of labeled emails, which are correctly marked as spam or non-spam, to predict whether a new email belongs to either of the two categories. A supervised learning task with discrete class labels, such as in the previous email spam filtering example, is also called a **classification task**. Another subcategory of supervised learning is **regression**, where the outcome signal is a continuous value.

Classification for predicting class labels

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances, based on past observations. Those class labels are discrete, unordered values that can be understood as the group memberships of the instances. The previously mentioned example of email spam detection represents a typical example of a binary classification task, where the machine learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam emails.

The following figure illustrates the concept of a binary classification task given 30 training examples; 15 training examples are labeled as the negative class (minus signs) and 15 training examples are labeled as the positive class (plus signs). In this scenario, our dataset is two-dimensional, which means that each example has two values associated with it: x_1 and x_2. Now, we can use a supervised machine learning algorithm to learn a rule—the decision boundary represented as a dashed line—that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values:



However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled instance.

A typical example of a **multiclass classification** task is handwritten character recognition. We can collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. The letters ("A," "B," "C," and so on) will represent the different unordered categories or class labels that we want to predict. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with certain accuracy. However, our machine learning system will be unable to correctly recognize any of the digits between 0 and 9, for example, if they were not part of the training dataset.

Regression for predicting continuous outcomes

We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called **regression analysis**. In regression analysis, we are given a number of predictor (**explanatory**) variables and a continuous response variable (**outcome**), and we try to find a relationship between those variables that allows us to predict an outcome.

Note that in the field of machine learning, the predictor variables are commonly called "features," and the response variables are usually referred to as "target variables." We will adopt these conventions throughout this book.

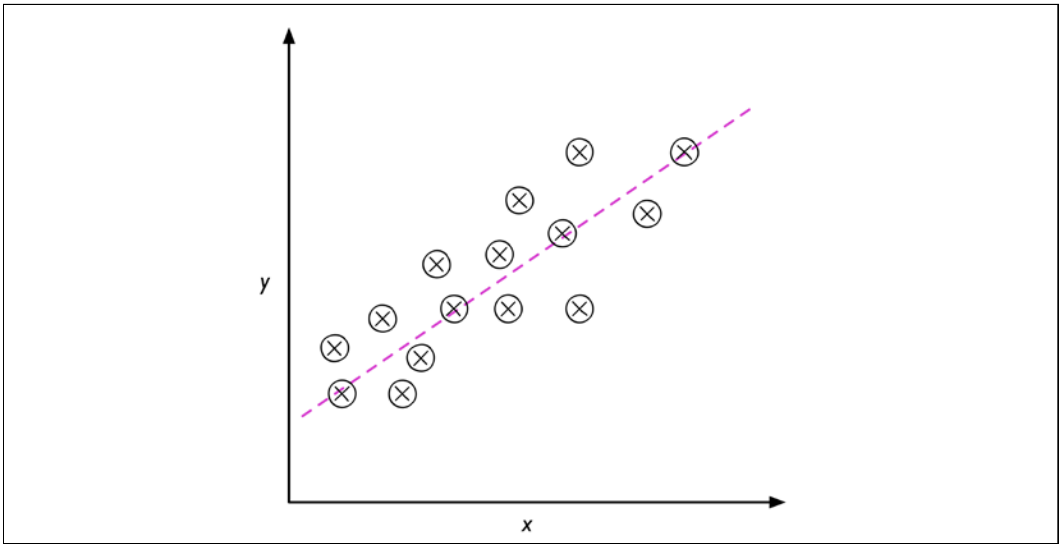
For example, let's assume that we are interested in predicting the math SAT scores of students. If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses the study time to predict the test scores of future students who are planning to take this test.

Regression toward the mean

The term "regression" was devised by Francis Galton in his article *Regression towards Mediocrity in Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of height in a population does not increase over time.

He observed that the height of parents is not passed on to their children, but instead, their children's height regresses toward the population mean.

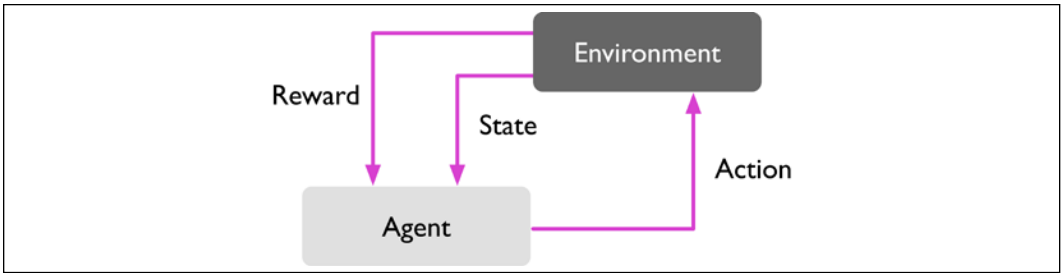
The following figure illustrates the concept of linear regression. Given a feature variable, x, and a target variable, y, we fit a straight line to this data that minimizes the distance—most commonly the average squared distance—between the data points and the fitted line. We can now use the intercept and slope learned from this data to predict the target variable of new data:



Solving interactive problems with reinforcement learning

Another type of machine learning is **reinforcement learning**. In reinforcement learning, the goal is to develop a system (**agent**) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called **reward signal**, we can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning, this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through its interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as **win** or **lose** at the end of the game:



There are many different subtypes of reinforcement learning. However, a general scheme is that the agent in reinforcement learning tries to maximize the reward through a series of interactions with the environment. Each state can be associated with a positive or negative reward, and a reward can be defined as accomplishing an overall goal, such as winning or losing a game of chess. For instance, in chess, the outcome of each move can be thought of as a different state of the environment.

To explore the chess example further, let's think of visiting certain configurations on the chess board as being associated with states that will more likely lead to winning—for instance, removing an opponent's chess piece from the board or threatening the queen. Other positions, however, are associated with states that will more likely result in losing the game, such as losing a chess piece to the opponent in the following turn. Now, in the game of chess, the reward (either positive for winning or negative for losing the game) will not be given until the end of the game. In addition, the final reward will also depend on how the opponent plays. For example, the opponent may sacrifice the queen but eventually win the game.

Reinforcement learning is concerned with learning to choose a series of actions that maximizes the total reward, which could be earned either immediately after taking an action or via *delayed* feedback.

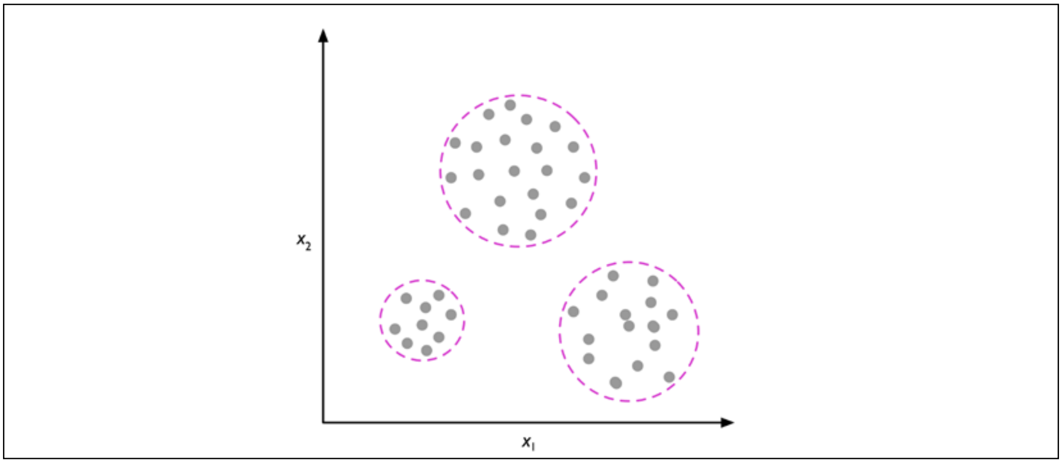
Discovering hidden structures with unsupervised learning

In supervised learning, we know the right answer beforehand when we train a model, and in reinforcement learning, we define a measure of reward for particular actions carried out by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function.

Finding subgroups with clustering

Clustering is an exploratory data analysis technique that allows us to organize a pile of information into meaningful subgroups (**clusters**) without having any prior knowledge of their group memberships. Each cluster that arises during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called **unsupervised classification**. Clustering is a great technique for structuring information and deriving meaningful relationships from data. For example, it allows marketers to discover customer groups based on their interests, in order to develop distinct marketing programs.

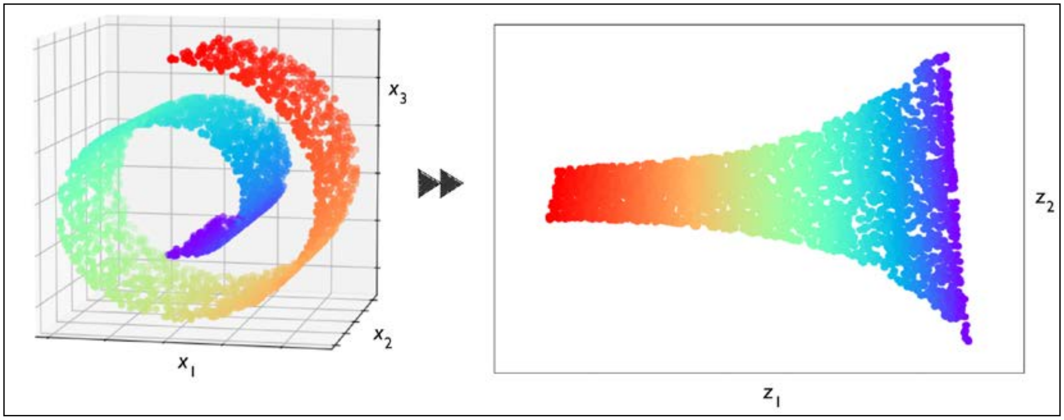
The following figure illustrates how clustering can be applied to organizing unlabeled data into three distinct groups based on the similarity of their features, x_1 and x_2:



Dimensionality reduction for data compression

Another subfield of unsupervised learning is **dimensionality reduction**. Often, we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of machine learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can also degrade the predictive performance of certain algorithms, and compress the data onto a smaller dimensional subspace while retaining most of the relevant information.

Sometimes, dimensionality reduction can also be useful for visualizing data; for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces in order to visualize it via 2D or 3D scatterplots or histograms. The following figure shows an example where nonlinear dimensionality reduction was applied to compress a 3D Swiss Roll onto a new 2D feature subspace:



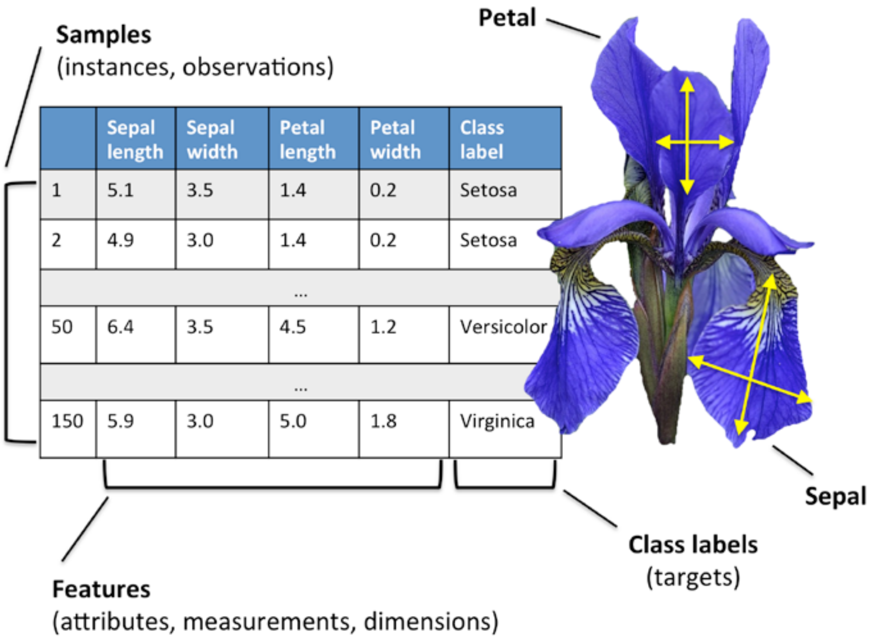
Introduction to the basic terminology and notations

Now that we have discussed the three broad categories of machine learning—supervised, unsupervised, and reinforcement learning—let's have a look at the basic terminology that we will be using throughout this book. The following subsection covers the common terms we will be using when referring to different aspects of a dataset, as well as the mathematical notation to communicate more precisely and efficiently.

As machine learning is a vast field and very interdisciplinary, you are guaranteed to encounter many different terms that refer to the same concepts sooner rather than later. The second subsection collects many of the most commonly used terms that are found in machine learning literature, which may be useful to you as a reference section when reading more diverse machine learning literature.

Notation and conventions used in this book

The following table depicts an excerpt of the Iris dataset, which is a classic example in the field of machine learning. The Iris dataset contains the measurements of 150 Iris flowers from three different species—Setosa, Versicolor, and Virginica. Here, each flower example represents one row in our dataset, and the flower measurements in centimeters are stored as columns, which we also call the **features** of the dataset:



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of linear algebra. In the following chapters, we will use a matrix and vector notation to refer to our data. We will follow the common convention to represent each example as a separate row in a feature matrix, **X**, where each feature is stored as a separate column.

The Iris dataset, consisting of 150 examples and four features, can then be written as a 150×4 matrix, $\boldsymbol{X} \in \mathbb{R}^{\mathbf{150 \times 4}}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

Notational conventions

For the rest of this book, unless noted otherwise, we will use the superscript *i* to refer to the *i*th training example, and the subscript *j* to refer to the *j*th dimension of the training dataset.

We will use lowercase, bold-face letters to refer to vectors ($\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{n \times 1}}$) and uppercase, bold-face letters to refer to matrices ($\boldsymbol{X} \in \mathbb{R}^{\boldsymbol{n \times m}}$). To refer to single elements in a vector or matrix, we will write the letters in italics ($x^{(n)}$ or $x_m^{(n)}$, respectively).

For example, $x_1^{(150)}$ refers to the first dimension of flower example 150, the *sepal length*. Thus, each row in this feature matrix represents one flower instance and can be written as a four-dimensional row vector, $\boldsymbol{x}^{(i)} \in \mathbb{R}^{\mathbf{1 \times 4}}$:

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

And each feature dimension is a 150-dimensional column vector, $\boldsymbol{x}^{(i)} \in \mathbb{R}^{\mathbf{150 \times 1}}$. For example:

$$\boldsymbol{x_j} = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \dots \\ x_j^{(150)} \end{bmatrix}$$

Similarly, we will store the target variables (here, class labels) as a 150-dimensional column vector:

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} (\mathrm{y} \in \{\text{Setosa}, \text{Versicolor}, \text{Virginica}\})$$

Machine learning terminology

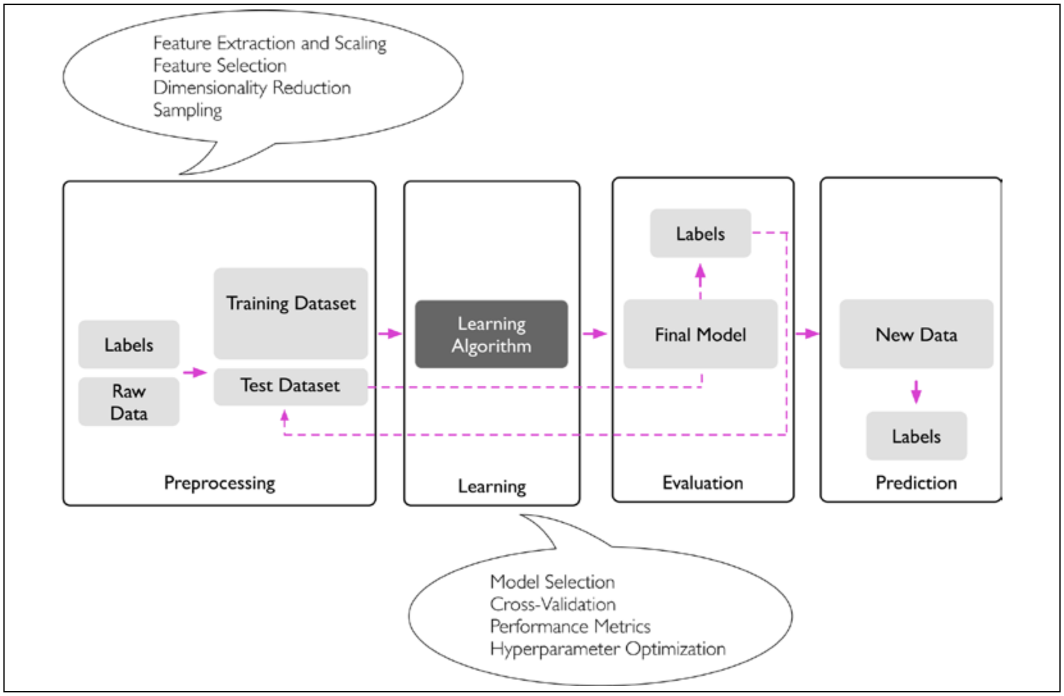
Machine learning is a vast field and also very interdisciplinary as it brings together many scientists from other areas of research. As it happens, many terms and concepts have been rediscovered or redefined and may already be familiar to you but appear under different names. For your convenience, in the following list, you can find a selection of commonly used terms and their synonyms that you may find useful when reading this book and machine learning literature in general:

- Training example: A row in a table representing the dataset and synonymous with an observation, record, instance, or sample (in most contexts, sample refers to a collection of training examples).
- Training: Model fitting, for parametric models similar to parameter estimation.
- Feature, abbrev. x: A column in a data table or data (design) matrix. Synonymous with predictor, variable, input, attribute, or covariate.
- Target, abbrev. y: Synonymous with outcome, output, response variable, dependent variable, (class) label, and ground truth.
- Loss function: Often used synonymously with a *cost* function. Sometimes the loss function is also called an *error* function. In some literature, the term "loss" refers to the loss measured for a single data point, and the cost is a measurement that computes the loss (average or summed) over the entire dataset.

A roadmap for building machine learning systems

In previous sections, we discussed the basic concepts of machine learning and the three different types of learning. In this section, we will discuss the other important parts of a machine learning system accompanying the learning algorithm.

The following diagram shows a typical workflow for using machine learning in predictive modeling, which we will discuss in the following subsections:



Preprocessing – getting data into shape

Let's begin with discussing the roadmap for building machine learning systems. Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing of the data is one of the most crucial steps in any machine learning application.

If we take the Iris flower dataset from the previous section as an example, we can think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be the color, hue, and intensity of the flowers, or the height, length, and width of the flowers.

Many machine learning algorithms also require that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range [0, 1] or a standard normal distribution with zero mean and unit variance, as we will see in later chapters.

Some of the selected features may be highly correlated and therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the advantage that less storage space is required, and the learning algorithm can run much faster. In certain cases, dimensionality reduction can also improve the predictive performance of a model if the dataset contains a large number of irrelevant features (or noise); that is, if the dataset has a low signal-to-noise ratio.

To determine whether our machine learning algorithm not only performs well on the training dataset but also generalizes well to new data, we also want to randomly divide the dataset into a separate training and test dataset. We use the training dataset to train and optimize our machine learning model, while we keep the test dataset until the very end to evaluate the final model.

Training and selecting a predictive model

As you will see in later chapters, many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous *No free lunch theorems* is that we can't get learning "for free" (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert, 1996; *No free lunch theorems for optimization*, D.H. Wolpert and W.G. Macready, 1997). We can relate this concept to the popular saying, "*I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail*" (Abraham Maslow, 1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority if we don't make any assumptions about the task. In practice, it is therefore essential to compare at least a handful of different algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is this: *how do we know which model performs well on the final test dataset and real-world data if we don't use this test dataset for the model selection, but keep it for the final model evaluation?* In order to address the issue embedded in this question, different techniques summarized as "cross-validation" can be used. In cross-validation, we further divide a dataset into training and validation subsets in order to estimate the generalization performance of the model. Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter optimization techniques that help us to fine-tune the performance of our model in later chapters.

We can think of those hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve its performance. This will become much clearer in later chapters when we see actual examples.

Evaluating models and predicting unseen data instances

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the so-called generalization error. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned procedures, such as feature scaling and dimensionality reduction, are solely obtained from the training dataset, and the same parameters are later reapplied to transform the test dataset, as well as any new data instances—the performance measured on the test data may be overly optimistic otherwise.

Using Python for machine learning

Python is one of the most popular programming languages for data science and thanks to its very active developer and open source community, a large number of useful libraries for scientific computing and machine learning have been developed.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as NumPy and SciPy have been developed that build upon lower-layer Fortran and C implementations for fast vectorized operations on multidimensional arrays.

For machine learning programming tasks, we will mostly refer to the scikit-learn library, which is currently one of the most popular and accessible open source machine learning libraries. In the later chapters, when we focus on a subfield of machine learning called deep learning, we will use the latest version of the TensorFlow library, which specializes in training so-called deep neural network models very efficiently by utilizing graphics cards.

Installing Python and packages from the Python Package Index

Python is available for all three major operating systems—Microsoft Windows, macOS, and Linux—and the installer, as well as the documentation, can be downloaded from the official Python website: `https://www.python.org`.

This book is written for Python version 3.7 or higher, and it is recommended that you use the most recent version of Python 3 that is currently available. Some of the code may also be compatible with Python 2.7, but as the official support for Python 2.7 ends in 2019, and the majority of open source libraries have already stopped supporting Python 2.7 (`https://python3statement.org`), we strongly advise that you use Python 3.7 or newer.

The additional packages that we will be using throughout this book can be installed via the `pip` installer program, which has been part of the Python Standard Library since Python 3.3. More information about `pip` can be found at `https://docs.python.org/3/installing/index.html`.

After we have successfully installed Python, we can execute pip from the terminal to install additional Python packages:

```
pip install SomePackage
```

Already installed packages can be updated via the `--upgrade` flag:

```
pip install SomePackage --upgrade
```

Using the Anaconda Python distribution and package manager

A highly recommended alternative Python distribution for scientific computing is Anaconda by Continuum Analytics. Anaconda is a free—including commercial use—enterprise-ready Python distribution that bundles all the essential Python packages for data science, math, and engineering into one user-friendly, cross-platform distribution. The Anaconda installer can be downloaded at `https://docs.anaconda.com/anaconda/install/`, and an Anaconda quick start guide is available at `https://docs.anaconda.com/anaconda/user-guide/getting-started/`.

After successfully installing Anaconda, we can install new Python packages using the following command:

```
conda install SomePackage
```

Existing packages can be updated using the following command:

```
conda update SomePackage
```

Packages for scientific computing, data science, and machine learning

Throughout this book, we will mainly use NumPy's multidimensional arrays to store and manipulate data. Occasionally, we will make use of pandas, which is a library built on top of NumPy that provides additional higher-level data manipulation tools that make working with tabular data even more convenient. To augment your learning experience and visualize quantitative data, which is often extremely useful to make sense of it, we will use the very customizable Matplotlib library.

The version numbers of the major Python packages that were used to write this book are mentioned in the following list. Please make sure that the version numbers of your installed packages are equal to, or greater than, these version numbers to ensure that the code examples run correctly:

- NumPy 1.17.4
- SciPy 1.3.1
- scikit-learn 0.22.0
- Matplotlib 3.1.0
- pandas 0.25.3

Summary

In this chapter, we explored machine learning at a very high level and familiarized ourselves with the big picture and major concepts that we are going to explore in the following chapters in more detail. We learned that supervised learning is composed of two important subfields: classification and regression. While classification models allow us to categorize objects into known classes, we can use regression analysis to predict the continuous outcomes of target variables. Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.

We briefly went over the typical roadmap for applying machine learning to problem tasks, which we will use as a foundation for deeper discussions and hands-on examples in the following chapters. Finally, we set up our Python environment and installed and updated the required packages to get ready to see machine learning in action.

Later in this book, in addition to machine learning itself, we will introduce different techniques to preprocess a dataset, which will help you to get the best performance out of different machine learning algorithms. While we will cover classification algorithms quite extensively throughout the book, we will also explore different techniques for regression analysis and clustering.

We have an exciting journey ahead, covering many powerful techniques in the vast field of machine learning. However, we will approach machine learning one step at a time, building upon our knowledge gradually throughout the chapters of this book. In the following chapter, we will start this journey by implementing one of the earliest machine learning algorithms for classification, which will prepare us for *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, where we will cover more advanced machine learning algorithms using the scikit-learn open source machine learning library.

2

Training Simple Machine Learning Algorithms for Classification

In this chapter, we will make use of two of the first algorithmically described machine learning algorithms for classification: the perceptron and adaptive linear neurons. We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of machine learning algorithms for classification and how they can be efficiently implemented in Python.

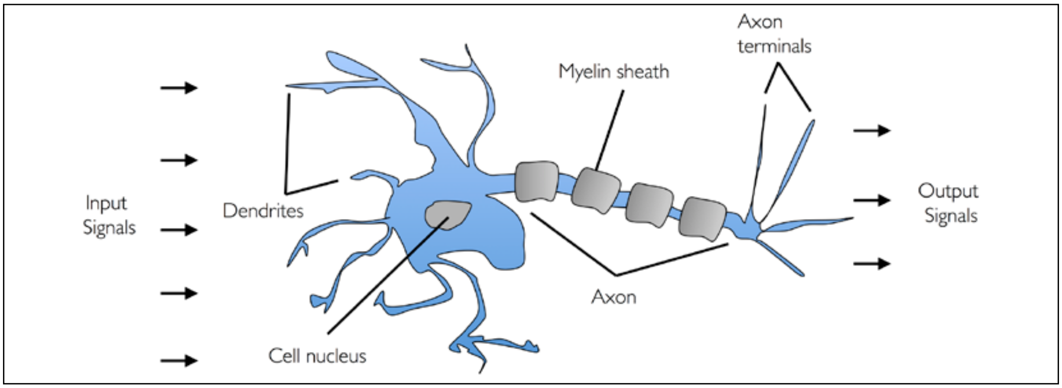
Discussing the basics of optimization using adaptive linear neurons will then lay the groundwork for using more sophisticated classifiers via the scikit-learn machine learning library in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*.

The topics that we will cover in this chapter are as follows:

- Building an understanding of machine learning algorithms
- Using pandas, NumPy, and Matplotlib to read in, process, and visualize data
- Implementing linear classification algorithms in Python

Artificial neurons – a brief glimpse into the early history of machine learning

Before we discuss the perceptron and related algorithms in more detail, let's take a brief tour of the beginnings of machine learning. Trying to understand how the biological brain works, in order to design artificial intelligence (AI), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called **McCulloch-Pitts** (**MCP**) neuron, in 1943 (*A Logical Calculus of the Ideas Immanent in Nervous Activity*, *W. S. McCulloch* and *W. Pitts*, *Bulletin of Mathematical Biophysics*, 5(4): 115-133, *1943*). Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:



McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton*, *F. Rosenblatt*, *Cornell Aeronautical Laboratory*, *1957*). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not. In the context of supervised learning and classification, such an algorithm could then be used to predict whether a new data point belongs to one class or the other.

The formal definition of an artificial neuron

More formally, we can put the idea behind **artificial neurons** into the context of a binary classification task where we refer to our two classes as *1* (positive class) and *–1* (negative class) for simplicity. We can then define a decision function ($\phi(z)$) that takes a linear combination of certain input values, ***x***, and a corresponding weight vector, ***w***, where *z* is the so-called net input $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m$:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Now, if the net input of a particular example, $\boldsymbol{x}^{(i)}$, is greater than a defined threshold, θ, we predict class *1*, and class *–1* otherwise. In the perceptron algorithm, the decision function, $\phi(\cdot)$, is a variant of a **unit step function**:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

For simplicity, we can bring the threshold, θ, to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$ so that we write *z* in a more compact form:

$$z = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$$

And:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

In machine learning literature, the negative threshold, or weight, $w_0 = -\theta$, is usually called the **bias unit**.

Linear algebra basics: dot product and matrix transpose

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\boldsymbol{x}$ and $\boldsymbol{w}$ using a vector dot product, whereas superscript T stands for transpose, which is an operation that transforms a column vector into a row vector and vice versa:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}$$

For example:

$$[1 \quad 2 \quad 3] \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

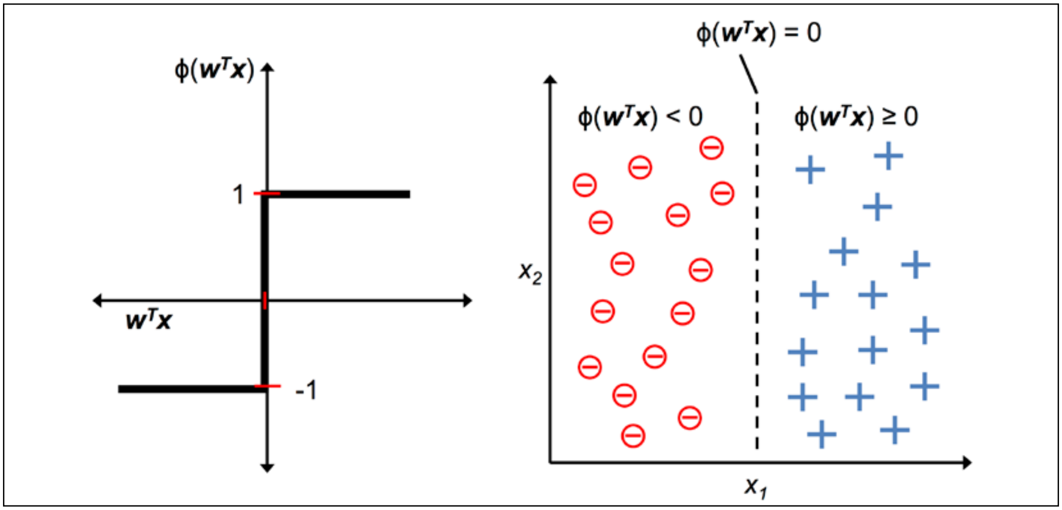
Furthermore, the transpose operation can also be applied to matrices to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Please note that the transpose operation is strictly only defined for matrices; however, in the context of machine learning, we refer to $n \times 1$ or $1 \times m$ matrices when we use the term "vector."

In this book, we will only use very basic concepts from linear algebra; however, if you need a quick refresher, please take a look at Zico Kolter's excellent *Linear Algebra Review and Reference*, which is freely available at `http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf`.

The following figure illustrates how the net input $z = \boldsymbol{w}^T \boldsymbol{x}$ is squashed into a binary output (–1 or 1) by the decision function of the perceptron (left subfigure) and how it can be used to discriminate between **two linearly separable classes** (right subfigure):


$\phi(\boldsymbol{w}^T\boldsymbol{x})$
1
$\boldsymbol{w}^T\boldsymbol{x}$
-1
$\phi(\boldsymbol{w}^T\boldsymbol{x}) = 0$
$\phi(\boldsymbol{w}^T\boldsymbol{x}) < 0$
$\phi(\boldsymbol{w}^T\boldsymbol{x}) \geq 0$
x_2
x_1

The perceptron learning rule

The whole idea behind the MCP neuron and Rosenblatt's *thresholded* perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either *fires* or it doesn't. Thus, Rosenblatt's initial perceptron rule is fairly simple, and the perceptron algorithm can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.
2. For each training example, $\boldsymbol{x}^{(i)}$:
    a. Compute the output value, $\hat{y}$.
    b. Update the weights.

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight, w_j, in the weight vector, $\boldsymbol{w}$, can be more formally written as:

$$w_j := w_j + \Delta w_j$$

The update value for w_j (or change in w_j), which we refer to as Δw_j, is calculated by the perceptron learning rule as follows:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$

Where η is the **learning rate** (typically a constant between 0.0 and 1.0), $y^{(i)}$ is the **true class label** of the *i*th training example, and $\hat{y}^{(i)}$ is the **predicted class label**. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the predicted label, $\hat{y}^{(i)}$, before all of the weights are updated via the respective update values, Δw_j. Concretely, for a two-dimensional dataset, we would write the update as:

$$\Delta w_0 = \eta\left(y^{(i)} - output^{(i)}\right)$$

$$\Delta w_1 = \eta\left(y^{(i)} - output^{(i)}\right)x_1^{(i)}$$

$$\Delta w_2 = \eta\left(y^{(i)} - output^{(i)}\right)x_2^{(i)}$$

Before we implement the perceptron rule in Python, let's go through a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged, since the update values are 0:

(1) $y^{(i)} = -1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta\left(-1 - (-1)\right)x_j^{(i)} = 0$

(2) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$

However, in the case of a wrong prediction, the weights are being pushed toward the direction of the positive or negative target class:

(3) $y^{(i)} = 1, \quad \hat{y}^{(i)} = -1, \quad \Delta w_j = \eta\left(1 - (-1)\right)x_j^{(i)} = \eta(2)x_j^{(i)}$

(4) $y^{(i)} = -1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}$

To get a better understanding of the multiplicative factor, $x_j^{(i)}$, let's go through another simple example, where:

$$\hat{y}^{(i)} = -1, \qquad y^{(i)} = +1, \qquad \eta = 1$$

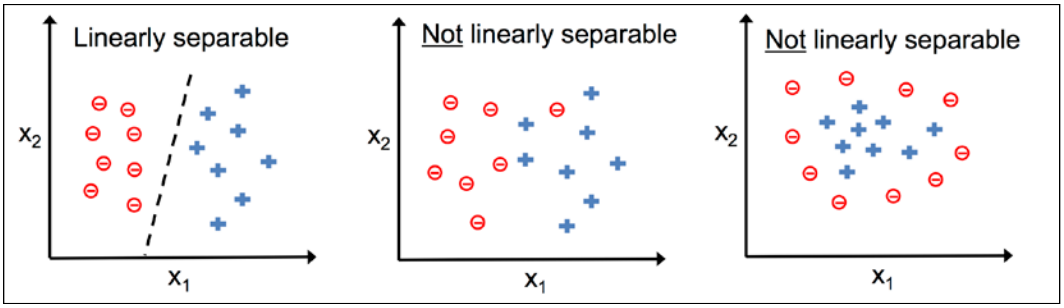
Let's assume that $x_j^{(i)} = 0.5$, and we misclassify this example as *–1*. In this case, we would increase the corresponding weight by 1 so that the net input, $x_j^{(i)} \times w_j$, would be more positive the next time we encounter this example, and thus be more likely to be above the threshold of the unit step function to classify the example as *+1*:

$$\Delta w_j = (1 - (-1))0.5 = (2)0.5 = 1$$

The weight update is proportional to the value of $x_j^{(i)}$. For instance, if we have another example, $x_j^{(i)} = 2$, that is incorrectly classified as *–1*, we will push the decision boundary by an even larger extent to classify this example correctly the next time:

$$\Delta w_j = \left(1^{(i)} - (-1)^{(i)}\right)2^{(i)} = (2)2^{(i)} = 4$$

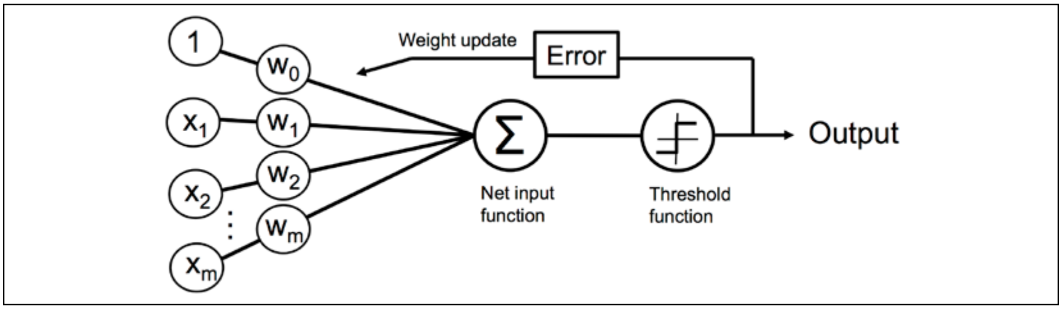
It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small (interested readers can find the mathematical proof in my lecture notes: `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf`.). If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (**epochs**) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise:



Downloading the example code

If you bought this book directly from Packt, you can download the example code files from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can download all code examples and datasets directly from `https://github.com/rasbt/python-machine-learning-book-3rd-edition`.

Now, before we jump into the implementation in the next section, what you just learned can be summarized in a simple diagram that illustrates the general concept of the perceptron:



The preceding diagram illustrates how the perceptron receives the inputs of an example, **x**, and combines them with the weights, **w**, to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of –1 or +1—the predicted class label of the example. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's now implement it in Python and apply it to the Iris dataset that we introduced in *Chapter 1*, *Giving Computers the Ability to Learn from Data*.

An object-oriented perceptron API

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new `Perceptron` objects that can learn from data via a `fit` method, and make predictions via a separate `predict` method. As a convention, we append an underscore (_) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

Additional resources for Python's scientific computing stack

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

- **NumPy**: `https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf`
- **pandas**: `https://pandas.pydata.org/pandas-docs/stable/10min.html`
- **Matplotlib**: `https://matplotlib.org/tutorials/introductory/usage.html`

The following is the implementation of a perceptron in Python:

```
import numpy as np

class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    errors_ : list
      Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
```

```
            self.random_state = random_state

        def fit(self, X, y):
            """Fit training data.

            Parameters
            ----------
            X : {array-like}, shape = [n_examples, n_features]
              Training vectors, where n_examples is the number of
              examples and n_features is the number of features.
            y : array-like, shape = [n_examples]
              Target values.

            Returns
            -------
            self : object

            """
            rgen = np.random.RandomState(self.random_state)
            self.w_ = rgen.normal(loc=0.0, scale=0.01,
                                  size=1 + X.shape[1])
            self.errors_ = []

            for _ in range(self.n_iter):
                errors = 0
                for xi, target in zip(X, y):
                    update = self.eta * (target - self.predict(xi))
                    self.w_[1:] += update * xi
                    self.w_[0] += update
                    errors += int(update != 0.0)
                self.errors_.append(errors)
            return self

        def net_input(self, X):
            """Calculate net input"""
            return np.dot(X, self.w_[1:]) + self.w_[0]

        def predict(self, X):
            """Return class label after unit step"""
            return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate, `eta`, and the number of epochs, `n_iter` (passes over the training dataset).

Via the `fit` method, we initialize the weights in `self.w_` to a vector, $\mathbb{R}^{m+1}$, where m stands for the number of dimensions (features) in the dataset, and we add *1* for the first element in this vector that represents the bias unit. Remember that the first element in this vector, `self.w_[0]`, represents the so-called bias unit that we discussed earlier.

Also notice that this vector contains small random numbers drawn from a normal distribution with standard deviation 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

It is important to keep in mind that we don't initialize the weights to zero because the learning rate, η (`eta`), only has an effect on the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter, `eta`, affects only the scale of the weight vector, not the direction. If you are familiar with trigonometry, consider a vector, $v1 = [1 \quad 2 \quad 3]$, where the angle between $v1$ and a vector, $v2 = 0.5 \times v1$, would be exactly zero, as demonstrated by the following code snippet:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...            np.linalg.norm(v2)))
0.0
```

Here, `np.arccos` is the trigonometric inverse cosine, and `np.linalg.norm` is a function that computes the length of a vector (our decision to draw the random numbers from a random normal distribution—for example, instead of from a uniform distribution—and to use a standard deviation of `0.01` was arbitrary; remember, we are just interested in small random values to avoid the properties of all-zero vectors, as discussed earlier).

NumPy array indexing

NumPy indexing for one-dimensional arrays works similarly to Python lists using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a two-dimensional array, `X`.

After the weights have been initialized, the `fit` method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule that we discussed in the previous section.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $\boldsymbol{w}^T\boldsymbol{x}$.

> Instead of using NumPy to calculate the vector dot product between two arrays, `a` and `b`, via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i * j for i, j in zip(a, b)])`. However, the advantage of using NumPy over classic Python for loop structures is that its arithmetic operations are vectorized. **Vectorization** means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array, rather than performing a set of operations for each element at a time, we can make better use of our modern central processing unit (CPU) architectures with **single instruction, multiple data** (**SIMD**) support. Furthermore, NumPy uses highly optimized linear algebra libraries, such as **Basic Linear Algebra Subprograms** (**BLAS**) and **Linear Algebra Package** (**LAPACK**), that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this chapter to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, sepal length and petal length, will allow us to visualize the decision regions of the trained model in a scatter plot for learning purposes.

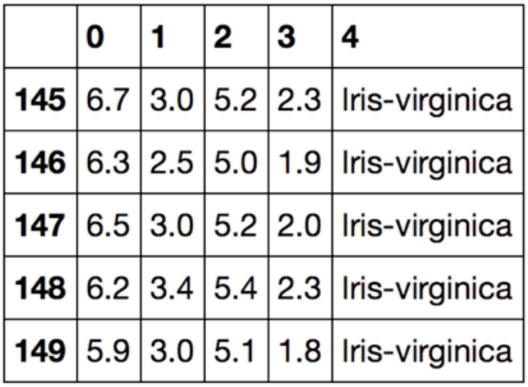
Note that we will also only consider two flower classes, Setosa and Versicolor, from the Iris dataset for practical reasons—remember, the perceptron is a binary classifier. However, the perceptron algorithm can be extended to multi-class classification—for example, the **one-vs.-all** (**OvA**) technique.

The OvA method for multi-class classification

OvA, which is sometimes also called **one-vs.-rest** (**OvR**), is a technique that allows us to extend any binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the examples from all other classes are considered negative classes. If we were to classify a new, unlabeled data instance, we would use our *n* classifiers, where *n* is the number of class labels, and assign the class label with the highest confidence to the particular instance we want to classify. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

First, we will use the `pandas` library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a `DataFrame` object and print the last five lines via the `tail` method to check that the data was loaded correctly:

```
>>> import os
>>> import pandas as pd
>>> s = os.path.join('https://archive.ics.uci.edu', 'ml',
...                  'machine-learning-databases',
...                  'iris','iris.data')
>>> print('URL:', s)
URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
iris.data
>>> df = pd.read_csv(s,
...                  header=None,
...                  encoding='utf-8')
>>> df.tail()
```



	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Loading the Iris dataset

You can find a copy of the Iris dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data` is temporarily unavailable. For instance, to load the Iris dataset from a local directory, you can replace this line,

```
df = pd.read_csv(
  'https://archive.ics.uci.edu/ml/'
  'machine-learning-databases/iris/iris.data',
  header=None, encoding='utf-8')
```

with the following one:

```
df = pd.read_csv(
  'your/local/path/to/iris.data',
  header=None, encoding='utf-8')
```

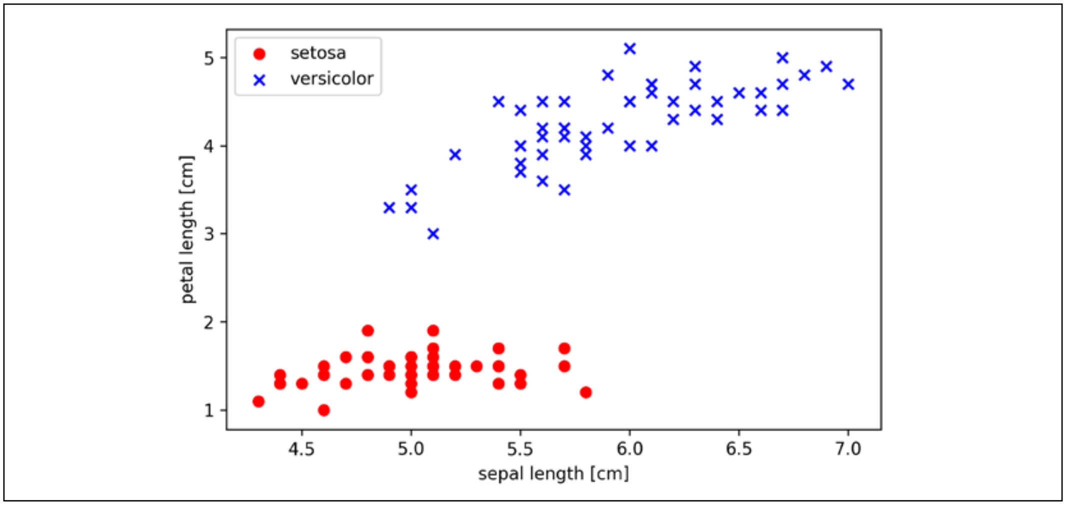
Next, we extract the first 100 class labels that correspond to the 50 Iris-setosa and 50 Iris-versicolor flowers and convert the class labels into the two integer class labels, `1` (versicolor) and `-1` (setosa), that we assign to a vector, `y`, where the `values` method of a pandas `DataFrame` yields the corresponding NumPy representation.

Similarly, we extract the first feature column (sepal length) and the third feature column (petal length) of those 100 training examples and assign them to a feature matrix, `X`, which we can visualize via a two-dimensional scatterplot:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> # select setosa and versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)

>>> # extract sepal length and petal length
>>> X = df.iloc[0:100, [0, 2]].values

>>> # plot data
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='x', label='versicolor')
```

```
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see the following scatterplot:



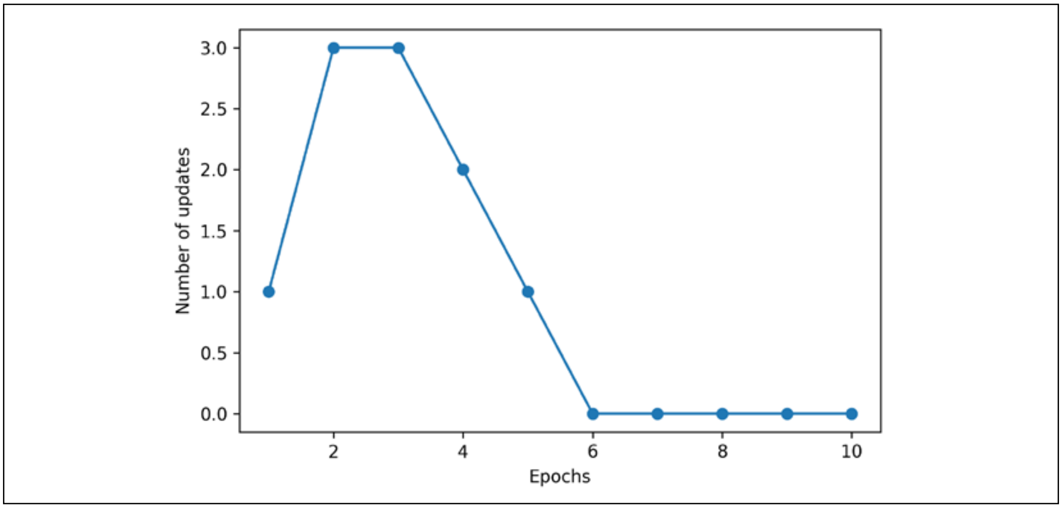
The preceding scatterplot shows the distribution of flower examples in the Iris dataset along the two feature axes: petal length and sepal length (measured in centimeters). In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate Setosa from Versicolor flowers.

Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

Now, it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the misclassification error for each epoch to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
...          ppn.errors_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of updates')
>>> plt.show()
```

After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown in the following graph:



As we can see in the preceding plot, our perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly. Let's implement a small convenience function to visualize the decision boundaries for two-dimensional datasets:

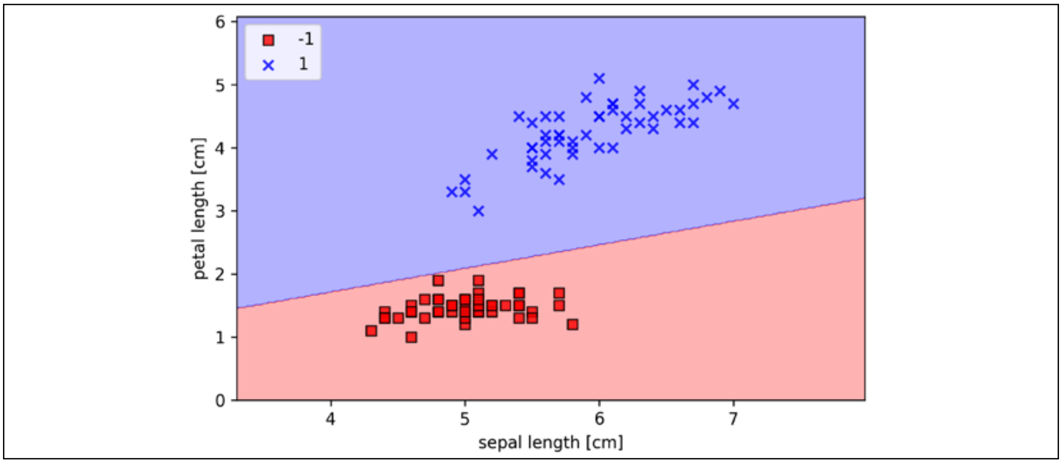
```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                     y=X[y == cl, 1],
                     alpha=0.8,
                     c=colors[idx],
                     marker=markers[idx],
                     label=cl,
                     edgecolor='black')
```

First, we define a number of `colors` and `markers` and create a colormap from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays, `xx1` and `xx2`, via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels, `Z`, of the corresponding grid points.

After reshaping the predicted class labels, `Z`, into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via Matplotlib's `contourf` function, which maps the different decision regions to different colors for each predicted class in the grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('sepal length [cm]')
>>> plt.ylabel('petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in the following figure:



As we can see in the plot, the perceptron learned a decision boundary that is able to classify all flower examples in the Iris training subset perfectly.

Perceptron convergence

Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if the classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs. Interested readers can find a summary of the proof in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf`.

Adaptive linear neurons and the convergence of learning

In this section, we will take a look at another type of single-layer neural network (NN): **ADAptive LInear NEuron** (**Adaline**). Adaline was published by Bernard Widrow and his doctoral student Tedd Hoff only a few years after Rosenblatt's perceptron algorithm, and it can be considered an improvement on the latter (*An Adaptive "Adaline" Neuron Using Chemical "Memistors"*, *Technical Report Number 1553-2, B. Widrow and others*, *Stanford Electron Labs*, Stanford, CA, *October 1960*).

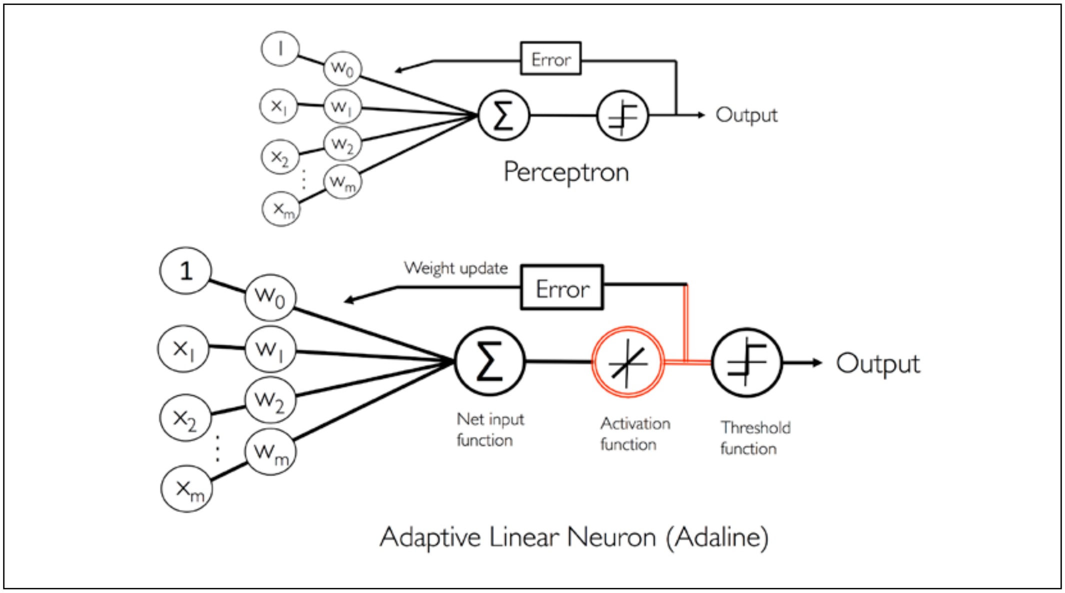
The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous cost functions. This lays the groundwork for understanding more advanced machine learning algorithms for classification, such as logistic regression, support vector machines, and regression models, which we will discuss in future chapters.

The key difference between the Adaline rule (also known as the **Widrow-Hoff rule**) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\phi(z)$, is simply the identity function of the net input, so that:

$$\phi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$$

While the linear activation function is used for learning the weights, we still use a threshold function to make the final prediction, which is similar to the unit step function that we covered earlier.

The main differences between the perceptron and Adaline algorithm are highlighted in the following figure:



As the illustration indicates, the Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

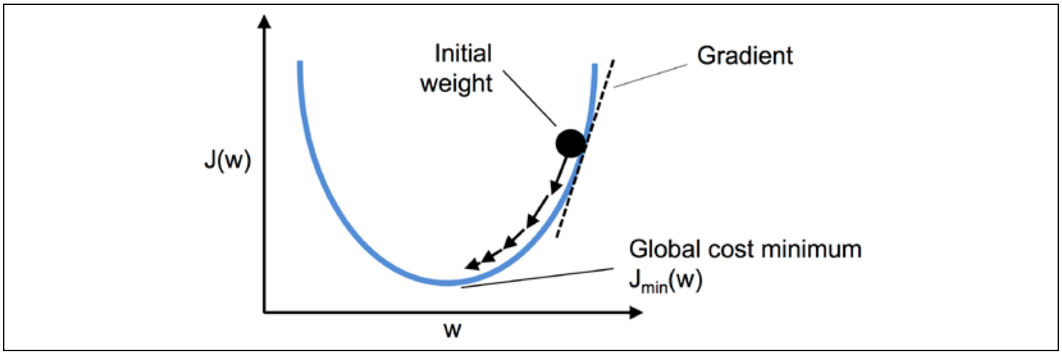
Minimizing cost functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is a defined **objective function** that is to be optimized during the learning process. This objective function is often a cost function that we want to minimize. In the case of Adaline, we can define the cost function, *J*, to learn the weights as the **sum of squared errors** (**SSE**) between the calculated outcome and the true class label:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

The term $\frac{1}{2}$ is just added for our convenience and will make it easier to derive the gradient of the cost or loss function with respect to the weight parameters, as we will see in the following paragraphs. The main advantage of this continuous linear activation function, in contrast to the unit step function, is that the cost function becomes differentiable. Another nice property of this cost function is that it is convex; thus, we can use a very simple yet powerful optimization algorithm called **gradient descent** to find the weights that minimize our cost function to classify the examples in the Iris dataset.

As illustrated in the following figure, we can describe the main idea behind gradient descent as *climbing down a hill* until a local or global cost minimum is reached. In each iteration, we take a step in the opposite direction of the gradient, where the step size is determined by the value of the learning rate, as well as the slope of the gradient:



Using gradient descent, we can now update the weights by taking a step in the opposite direction of the gradient, $\nabla J(\mathbf{w})$, of our cost function, $J(\mathbf{w})$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}$$

The weight change, $\Delta\boldsymbol{w}$, is defined as the negative gradient multiplied by the learning rate, η:

$$\Delta\boldsymbol{w} = -\eta\nabla J(\mathbf{w})$$

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight, w_j:

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

So we can write the update of weight w_j as:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Since we update all weights simultaneously, our Adaline learning rule becomes:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

The squared error derivative

If you are familiar with calculus, the partial derivative of the SSE cost function with respect to the *j*th weight can be obtained as follows:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

$$= \frac{1}{2} \sum_i 2\left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)}\right)\right)$$

$$= \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)\left(-x_j^{(i)}\right)$$

$$= -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\phi\left(z^{(i)}\right)$ with $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all examples in the training dataset (instead of updating the weights incrementally after each training example), which is why this approach is also referred to as **batch gradient descent**.

Implementing Adaline in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation that we defined earlier and change the `fit` method so that the weights are updated by minimizing the cost function via gradient descent:

```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples
            is the number of examples and
            n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X))
                        >= 0.0, 1, -1)
```

Instead of updating the weights after evaluating each individual training example, as in the perceptron, we calculate the gradient based on the whole training dataset via `self.eta * errors.sum()` for the bias unit (zero-weight), and via `self.eta * X.T.dot(errors)` for the weights 1 to *m*, where `X.T.dot(errors)` is a matrix-vector multiplication between our feature matrix and the error vector.

Please note that the `activation` method has no effect in the code since it is simply an identity function. Here, we added the activation function (computed via the `activation` method) to illustrate the general concept with regard to how information flows through a single-layer NN: features from the input data, net input, activation, and output.

In the next chapter, we will learn about a logistic regression classifier that uses a non-identity, nonlinear activation function. We will see that a logistic regression model is closely related to Adaline, with the only difference being its activation and cost function.

Now, similar to the previous perceptron implementation, we collect the cost values in a `self.cost_` list to check whether the algorithm converged after training.

Matrix multiplication

Performing a matrix multiplication is similar to calculating a vector dot-product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}7\\8\\9\end{bmatrix} = \begin{bmatrix}1\times 7 + \ 2\times 8 + \ 3\times 9\\4\times 7 + \ 5\times 8 + \ 6\times 9\end{bmatrix} = \begin{bmatrix}50\\122\end{bmatrix}$$

Please note that in the preceding equation, we are multiplying a matrix with a vector, which is mathematically not defined. However, remember that we use the convention that this preceding vector is regarded as a 3×1 matrix.

In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence. So, let's choose two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, to start with and plot the cost functions versus the number of epochs to see how well the Adaline implementation learns from the training data.

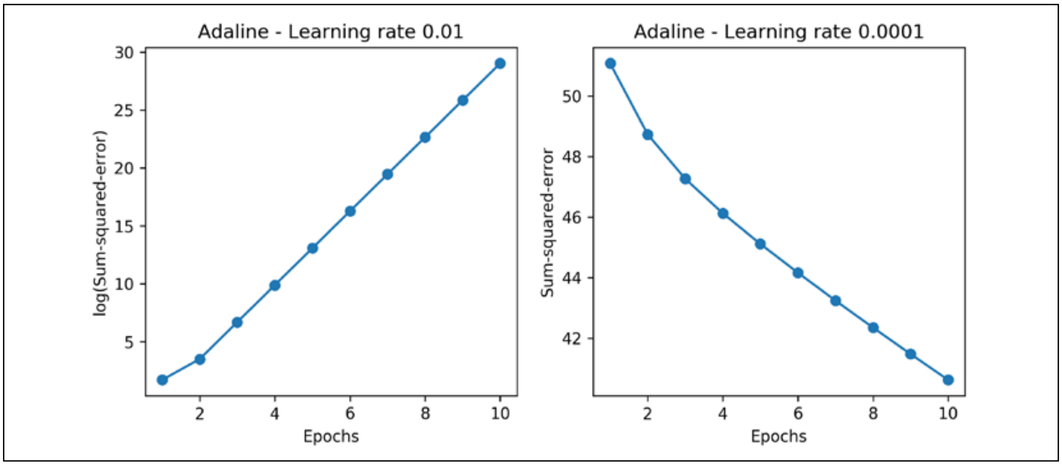
Perceptron hyperparameters

The learning rate, η, (`eta`), as well as the number of epochs (`n_iter`), are the so-called hyperparameters (or tuning parameters) of the perceptron and Adaline learning algorithms. In *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will take a look at different techniques to automatically find the values of different hyperparameters that yield optimal performance of the classification model.

Let's now plot the cost against the number of epochs for the two different learning rates:

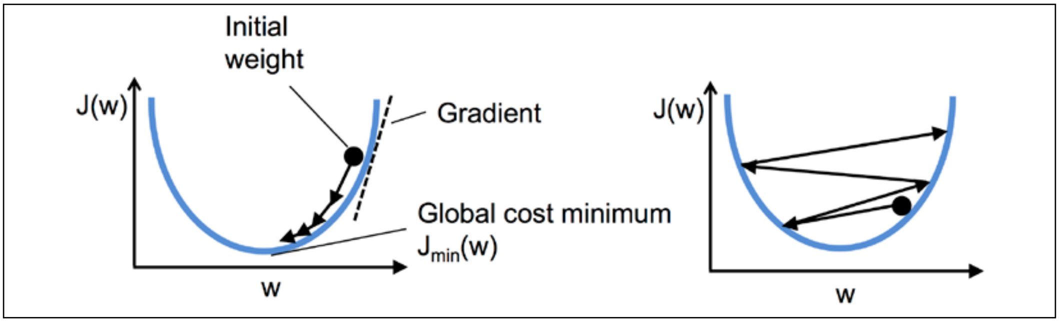
```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

>>> ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.cost_) + 1),
...            np.log10(ada1.cost_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Sum-squared-error)')
>>> ax[0].set_title('Adaline - Learning rate 0.01')

>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...            ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Sum-squared-error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As we can see in the resulting cost-function plots, we encountered two different types of problem. The left chart shows what could happen if we choose a learning rate that is too large. Instead of minimizing the cost function, the error becomes larger in every epoch, because we *overshoot* the global minimum. On the other hand, we can see that the cost decreases on the right plot, but the chosen learning rate, $\eta = 0.0001$, is so small that the algorithm would require a very large number of epochs to converge to the global cost minimum:



The following figure illustrates what might happen if we change the value of a particular weight parameter to minimize the cost function, *J*. The left subfigure illustrates the case of a well-chosen learning rate, where the cost decreases gradually, moving in the direction of the global minimum.

The subfigure on the right, however, illustrates what happens if we choose a learning rate that is too large—we overshoot the global minimum:



Improving gradient descent through feature scaling

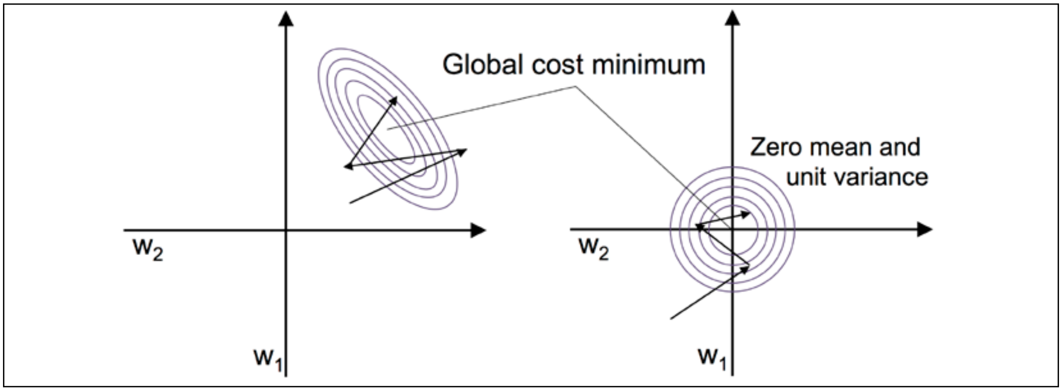
Many machine learning algorithms that we will encounter throughout this book require some sort of feature scaling for optimal performance, which we will discuss in more detail in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, and *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*.

Gradient descent is one of the many algorithms that benefit from feature scaling. In this section, we will use a feature scaling method called **standardization**, which gives our data the properties of a standard normal distribution: zero-mean and unit variance. This normalization procedure helps gradient descent learning to converge more quickly; however, it does not make the original dataset normally distributed. Standardization shifts the mean of each feature so that it is centered at zero and each feature has a standard deviation of 1 (unit variance). For instance, to standardize the *j*th feature, we can simply subtract the sample mean, μ_j, from every training example and divide it by its standard deviation, σ_j:

$$\boldsymbol{x}'_j = \frac{\boldsymbol{x}_j - \mu_j}{\sigma_j}$$

Here, $\boldsymbol{x}_j$ is a vector consisting of the *j*th feature values of all training examples, *n*, and this standardization technique is applied to each feature, *j*, in our dataset.

One of the reasons why standardization helps with gradient descent learning is that the optimizer has to go through fewer steps to find a good or optimal solution (the global cost minimum), as illustrated in the following figure, where the subfigures represent the cost surface as a function of two model weights in a two-dimensional classification problem:



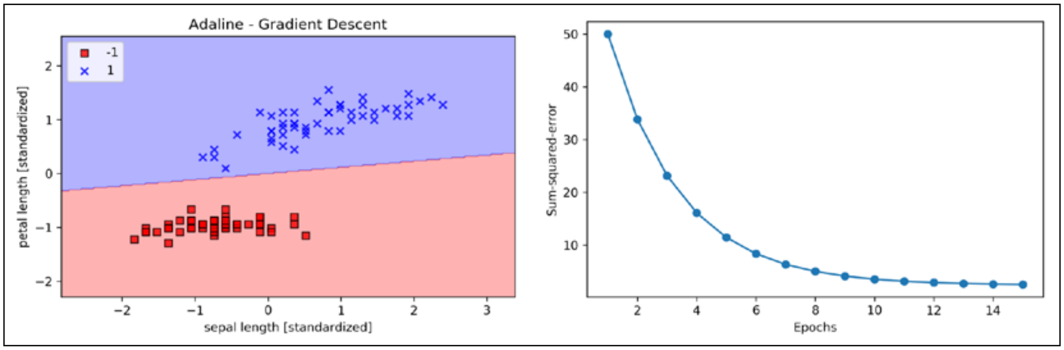
Standardization can easily be achieved by using the built-in NumPy methods `mean` and `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, we will train Adaline again and we will see that it now converges after a small number of epochs using a learning rate of $\eta = 0.01$:

```
>>> ada_gd = AdalineGD(n_iter=15, eta=0.01)
>>> ada_gd.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

>>> plt.plot(range(1, len(ada_gd.cost_) + 1),
...          ada_gd.cost_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Sum-squared-error')
>>> plt.tight_layout()
>>> plt.show()
```

After executing this code, we should see a figure of the decision regions, as well as a plot of the declining cost, as shown in the following figure:



As we can see in the plots, Adaline has now converged after training on the standardized features using a learning rate of $\eta = 0.01$. However, note that the SSE remains non-zero even though all flower examples were classified correctly.

Large-scale machine learning and stochastic gradient descent

In the previous section, we learned how to minimize a cost function by taking a step in the opposite direction of a cost gradient that is calculated from the whole training dataset; this is why this approach is sometimes also referred to as **batch gradient descent**. Now imagine that we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications. Running batch gradient descent can be computationally quite costly in such scenarios, since we need to reevaluate the whole training dataset each time that we take one step toward the global minimum.

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent (SGD)**, which is sometimes also called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\boldsymbol{x}^{(i)}$:

$$\Delta \boldsymbol{w} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \boldsymbol{x}^{(i)}$$

we update the weights incrementally for each training example:

$$\eta \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) \boldsymbol{x}^{(i)}$$

Although SGD can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that SGD can escape shallow local minima more readily if we are working with nonlinear cost functions, as we will see later in *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*. To obtain satisfying results via SGD, it is important to present training data in a random order; also, we want to shuffle the training dataset for every epoch to prevent cycles.

Adjusting the learning rate during training

In SGD implementations, the fixed learning rate, η, is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[number\ of\ iterations] + c_2}$$

where c_1 and c_2 are constants. Note that SGD does not reach the global minimum but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the cost minimum.

Another advantage of SGD is that we can use it for **online learning**. In online learning, our model is trained on the fly as new training data arrives. This is especially useful if we are accumulating large amounts of data, for example, customer data in web applications. Using online learning, the system can immediately adapt to changes, and the training data can be discarded after updating the model if storage space is an issue.

Mini-batch gradient descent

A compromise between batch gradient descent and SGD is so-called **mini-batch learning**. Mini-batch learning can be understood as applying batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time. The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the `for` loop over the training examples in SGD with vectorized operations leveraging concepts from linear algebra (for example, implementing a weighted sum via a dot product), which can further improve the computational efficiency of our learning algorithm.

Since we already implemented the Adaline learning rule using gradient descent, we only need to make a few adjustments to modify the learning algorithm to update the weights via SGD. Inside the `fit` method, we will now update the weights after each training example. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for online learning. In order to check whether our algorithm converged after training, we will calculate the cost as the average cost of the training examples in each epoch. Furthermore, we will add an option to shuffle the training data before each epoch to avoid repetitive cycles when we are optimizing the cost function; via the `random_state` parameter, we allow the specification of a random seed for reproducibility:

```
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent
        cycles.
    random_state : int
        Random number generator seed for random weight
        initialization.


    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value averaged over all
        training examples in each epoch.


    """
    def __init__(self, eta=0.01, n_iter=10,
              shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples is the number of
            examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.

        Returns
        -------
        self : object

        """
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
```

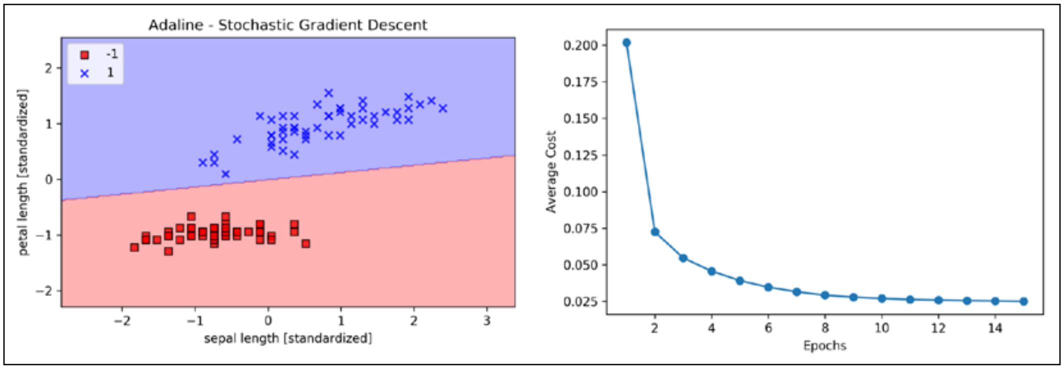
```
            return X[r], y[r]

        def _initialize_weights(self, m):
            """Initialize weights to small random numbers"""
            self.rgen = np.random.RandomState(self.random_state)
            self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
                                       size=1 + m)
            self.w_initialized = True

        def _update_weights(self, xi, target):
            """Apply Adaline learning rule to update the weights"""
            output = self.activation(self.net_input(xi))
            error = (target - output)
            self.w_[1:] += self.eta * xi.dot(error)
            self.w_[0] += self.eta * error
            cost = 0.5 * error**2
            return cost

        def net_input(self, X):
            """Calculate net input"""
            return np.dot(X, self.w_[1:]) + self.w_[0]

        def activation(self, X):
            """Compute linear activation"""
            return X

        def predict(self, X):
            """Return class label after unit step"""
            return np.where(self.activation(self.net_input(X))
                            >= 0.0, 1, -1)
```

The _shuffle method that we are now using in the AdalineSGD classifier works as follows: via the permutation function in np.random, we generate a random sequence of unique numbers in the range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix and class label vector.

We can then use the fit method to train the AdalineSGD classifier and use our plot_decision_regions to plot our training results:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)

>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - Stochastic Gradient Descent')
>>> plt.xlabel('sepal length [standardized]')
>>> plt.ylabel('petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.cost_) + 1), ada_sgd.cost_,
...          marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average Cost')
>>> plt.tight_layout()
>>> plt.show()
```

The two plots that we obtain from executing the preceding code example are shown in the following figure:



As you can see, the average cost goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent Adaline. If we want to update our model, for example, in an online learning scenario with streaming data, we could simply call the `partial_fit` method on individual training examples—for instance `ada_sgd.partial_fit(X_std[0, :], y[0])`.

Summary

In this chapter, we gained a good understanding of the basic concepts of linear classifiers for supervised learning. After we implemented a perceptron, we saw how we can train adaptive linear neurons efficiently via a vectorized implementation of gradient descent and online learning via SGD.

Now that we have seen how to implement simple classifiers in Python, we are ready to move on to the next chapter, where we will use the Python scikit-learn machine learning library to get access to more advanced and powerful machine learning classifiers, which are commonly used in academia as well as in industry.

The object-oriented approach that we used to implement the perceptron and Adaline algorithms will help with understanding the scikit-learn API, which is implemented based on the same core concepts that we used in this chapter: the `fit` and `predict` methods. Based on these core concepts, we will learn about logistic regression for modeling class probabilities and support vector machines for working with nonlinear decision boundaries. In addition, we will introduce a different class of supervised learning algorithms, tree-based algorithms, which are commonly combined into robust ensemble classifiers.

3

A Tour of Machine Learning Classifiers Using scikit-learn

In this chapter, we will take a tour of a selection of popular and powerful machine learning algorithms that are commonly used in academia as well as in industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an appreciation of their individual strengths and weaknesses. In addition, we will take our first steps with the scikit-learn library, which offers a user-friendly and consistent interface for using those algorithms efficiently and productively.

The topics that will be covered throughout this chapter are as follows:

- An introduction to robust and popular algorithms for classification, such as logistic regression, support vector machines, and decision trees
- Examples and explanations using the scikit-learn machine learning library, which provides a wide variety of machine learning algorithms via a user-friendly Python API
- Discussions about the strengths and weaknesses of classifiers with linear and nonlinear decision boundaries

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice and experience; each algorithm has its own quirks and is based on certain assumptions. To restate the **no free lunch theorem** by David H. Wolpert, no single classifier works best across all possible scenarios (*The Lack of A Priori Distinctions Between Learning Algorithms*, *Wolpert, David H*, *Neural Computation 8.7* (1996): 1341-1390). In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or examples, the amount of noise in a dataset, and whether the classes are linearly separable or not.

Eventually, the performance of a classifier—computational performance as well as predictive power—depends heavily on the underlying data that is available for learning. The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.
4. Evaluating the performance of the model.
5. Tuning the algorithm.

Since the approach of this book is to build machine learning knowledge step by step, we will mainly focus on the main concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in the book.

First steps with scikit-learn – training a perceptron

In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, you learned about two related learning algorithms for classification, the **perceptron** rule and **Adaline**, which we implemented in Python and NumPy by ourselves. Now we will take a look at the scikit-learn API, which, as mentioned, combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms. The scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. We will discuss this in more detail, together with the underlying concepts, in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in *Chapter 2*. For simplicity, we will use the already familiar **Iris dataset** throughout the following sections. Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Similar to the previous chapter, we will only use two features from the Iris dataset for visualization purposes.

We will assign the petal length and petal width of the 150 flower examples to the feature matrix, `X`, and the corresponding class labels of the flower species to the vector array, `y`:

```
>>> from sklearn import datasets
>>> import numpy as np

>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

The `np.unique(y)` function returned the three unique class labels stored in `iris.target`, and as we can see, the Iris flower class names, `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`, are already stored as integers (here: `0`, `1`, `2`). Although many scikit-learn functions and class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss the best practices around model evaluation in more detail:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y)
```

Using the `train_test_split` function from scikit-learn's `model_selection` module, we randomly split the `X` and `y` arrays into 30 percent test data (45 examples) and 70 percent training data (105 examples).

Note that the `train_test_split` function already shuffles the training datasets internally before splitting; otherwise, all examples from class `0` and class `1` would have ended up in the training datasets, and the test dataset would consist of 45 examples from class `2`. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset. We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
```

```
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we saw in the **gradient descent** example in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. Here, we will standardize the features using the `StandardScaler` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Using the preceding code, we loaded the `StandardScaler` class from the `preprocessing` module and initialized a new `StandardScaler` object that we assigned to the `sc` variable. Using the `fit` method, `StandardScaler` estimated the parameters, μ (sample mean) and σ (standard deviation), for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters, μ and σ. Note that we used the same scaling parameters to standardize the test dataset so that both the values in the training and test dataset are comparable to each other.

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the **one-vs.-rest** (**OvR**) method, which allows us to feed the three flower classes to the perceptron all at once. The code is as follows:

```
>>> from sklearn.linear_model import Perceptron

>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

The scikit-learn interface will remind you of our perceptron implementation in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. After loading the `Perceptron` class from the `linear_model` module, we initialized a new `Perceptron` object and trained the model via the `fit` method. Here, the model parameter, `eta0`, is equivalent to the learning rate, `eta`, that we used in our own perceptron implementation, and the `n_iter` parameter defines the number of epochs (passes over the training dataset).

As you will remember from *Chapter 2*, finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global cost minimum. If the learning rate is too small, the algorithm will require more epochs until convergence, which can make the learning slow—especially for large datasets. Also, we used the `random_state` parameter to ensure the reproducibility of the initial shuffling of the training dataset after each epoch.

Having trained a model in scikit-learn, we can make predictions via the `predict` method, just like in our own perceptron implementation in *Chapter 2*. The code is as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

Executing the code, we can see that the perceptron misclassifies 1 out of the 45 flower examples. Thus, the misclassification error on the test dataset is approximately 0.022 or 2.2 percent ($1/45 \approx 0.022$).

Classification error versus accuracy

Instead of the misclassification error, many machine learning practitioners report the classification accuracy of a model, which is simply calculated as follows:

1–*error* = 0.978 or 97.8 percent

Whether we use the classification error or accuracy is merely a matter of preference.

Note that scikit-learn also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test dataset as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Here, `y_test` are the true class labels and `y_pred` are the class labels that we predicted previously. Alternatively, each classifier in scikit-learn has a `score` method, which computes a classifier's prediction accuracy by combining the `predict` call with `accuracy_score`, as shown here:

```
>>> print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
Accuracy: 0.978
```

Overfitting

Note that we will evaluate the performance of our models based on the test dataset in this chapter. In *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, you will learn about useful techniques, including graphical analysis, such as learning curves, to detect and prevent **overfitting**. Overfitting, which we will return to later in this chapter, means that the model captures the patterns in the training data well but fails to generalize well to unseen data.

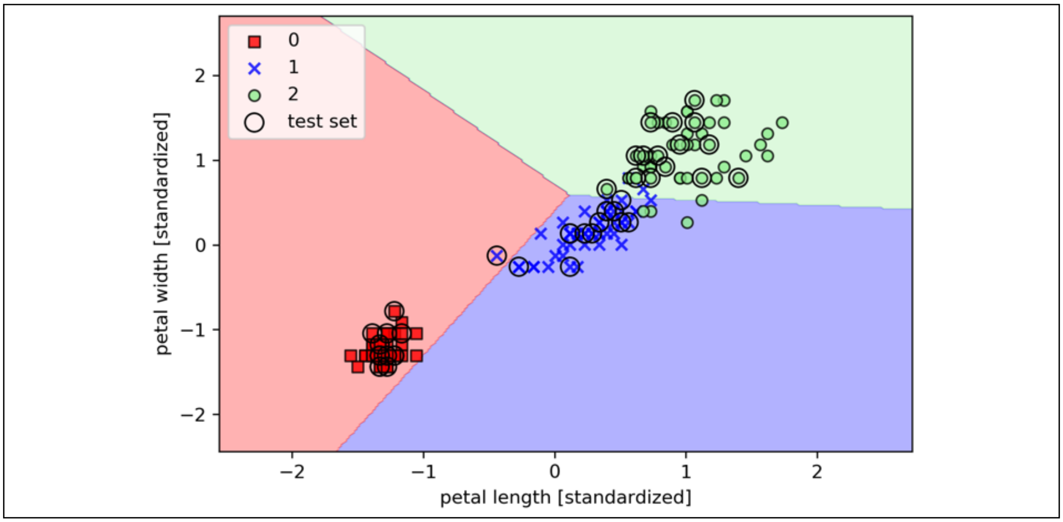
Finally, we can use our `plot_decision_regions` function from *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, to plot the **decision regions** of our newly trained perceptron model and visualize how well it separates the different flower examples. However, let's add a small modification to highlight the data instances from the test dataset via small circles:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None,
                          resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=colors[idx],
                    marker=markers[idx], label=cl,
                    edgecolor='black')
    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]

        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='test set')
```

With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the examples that we want to mark on the resulting plots. The code is as follows:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the three flower classes cannot be perfectly separated by a linear decision boundary:



Remember from our discussion in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice. In the following sections, we will look at more powerful linear classifiers that converge to a cost minimum even if the classes are not perfectly linearly separable.

Additional perceptron settings

The `Perceptron`, as well as other scikit-learn functions and classes, often has additional parameters that we omit for clarity. You can read more about those parameters using the `help` function in Python (for instance, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation at `http://scikit-learn.org/stable/`.

Modeling class probabilities via logistic regression

Although the perceptron rule offers a nice and easy-going introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. The reason for this is that the weights are continuously being updated, since there is always at least one misclassified training example present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset.

To make better use of our time, we will now take a look at another simple, yet more powerful, algorithm for linear and binary classification problems: **logistic regression**. Note that, in spite of its name, logistic regression is a model for classification, not regression.

Logistic regression and conditional probabilities

Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification.

Logistic regression for multiple classes

Note that logistic regression can be readily generalized to multiclass settings, which is known as multinomial logistic regression or softmax regression. A more detailed coverage of multinomial logistic regression is outside the scope of this book, but the interested reader can find more information in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L05_gradient-descent_slides.pdf` or `http://rasbt.github.io/mlxtend/user_guide/classifier/SoftmaxRegression/`.

Another way to use logistic regression in multiclass settings is via the OvR technique, which we discussed previously.

To explain the idea behind logistic regression as a probabilistic model for binary classification, let's first introduce the **odds**: the odds in favor of a particular event. The odds can be written as $\frac{p}{(1-p)}$ where p stands for the probability of the positive event. The term "positive event" does not necessarily mean "good," but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$. We can then further define the **logit** function, which is simply the logarithm of the odds (log-odds):

$$logit(p) = \log \frac{p}{(1-p)}$$

Note that *log* refers to the natural logarithm, as it is the common convention in computer science. The *logit* function takes input values in the range 0 to 1 and transforms them to values over the entire real-number range, which we can use to express a linear relationship between feature values and the log-odds:

$$logit\big(p(y=1|\boldsymbol{x})\big) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \boldsymbol{w}^T\boldsymbol{x}$$

Here, $p(y = 1|\boldsymbol{x})$ is the conditional probability that a particular example belongs to class 1 given its features, $\boldsymbol{x}$.

Now, we are actually interested in predicting the probability that a certain example belongs to a particular class, which is the inverse form of the logit function.

It is also called the **logistic sigmoid function**, which is sometimes simply abbreviated to **sigmoid function** due to its characteristic S-shape:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Here, *z* is the net input, the linear combination of weights, and the inputs (that is, the features associated with the training examples):

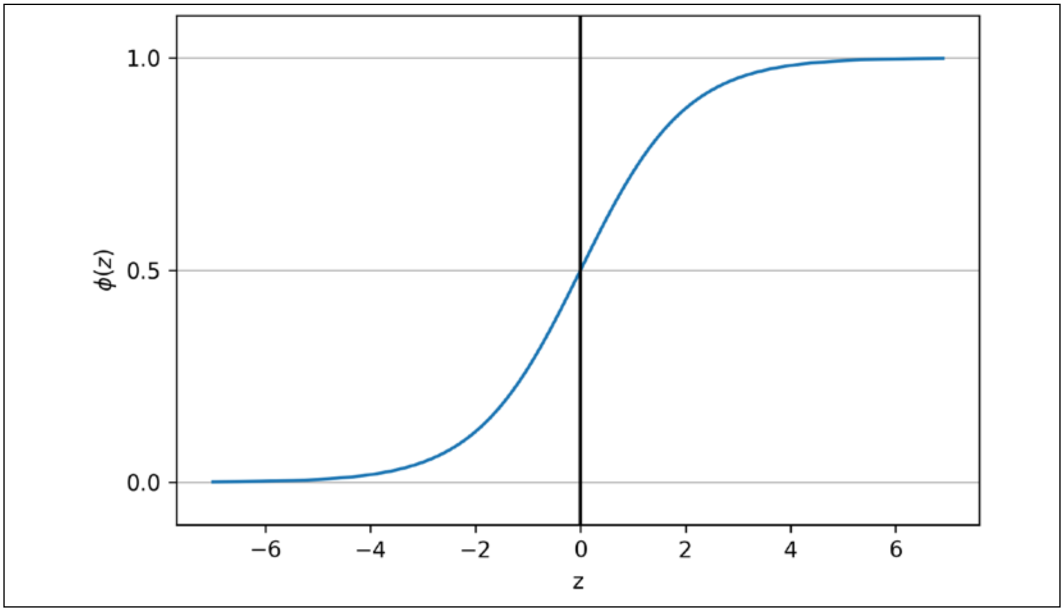
$$z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

Note that similar to the convention we used in *Chapter 2, Training Simple Machine Learning Algorithms for Classification,* w_0 refers to the bias unit and is an additional input value that we provide to x_0, which is set equal to 1.

Now, let's simply plot the sigmoid function for some values in the range –7 to 7 to see how it looks:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi (z)$')
>>> # y axis ticks and gridline
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

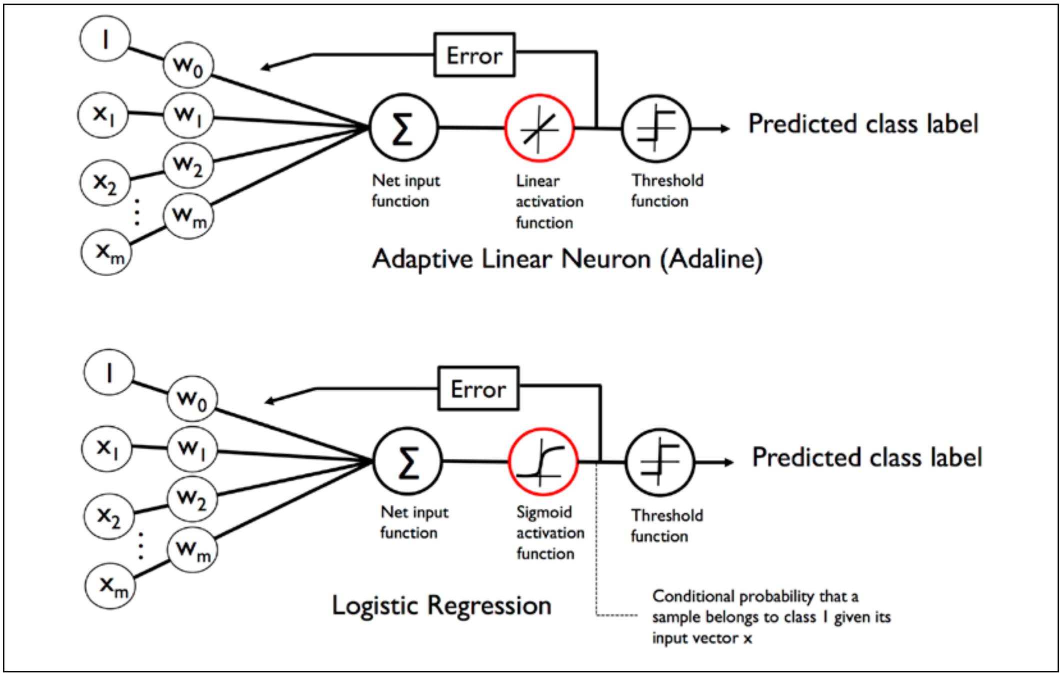
As a result of executing the previous code example, we should now see the S-shaped (sigmoidal) curve:



We can see that $\phi(z)$ approaches 1 if z goes toward infinity ($z \to \infty$) since e^{-z} becomes very small for large values of z. Similarly, $\phi(z)$ goes toward 0 for $z \to -\infty$ as a result of an increasingly large denominator. Thus, we can conclude that this sigmoid function takes real-number values as input and transforms them into values in the range [0, 1] with an intercept at $\phi(z) = 0.5$.

To build some understanding of the logistic regression model, we can relate it to *Chapter 2*. In Adaline, we used the identity function, $\phi(z) = z$, as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier.

The difference between Adaline and logistic regression is illustrated in the following figure:



The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\phi(z) = P(y = 1|\boldsymbol{x}; \boldsymbol{w})$, given its features, $\boldsymbol{x}$, parameterized by the weights, $\boldsymbol{w}$. For example, if we compute $\phi(z) = 0.8$ for a particular flower example, it means that the chance that this example is an `Iris-versicolor` flower is 80 percent. Therefore, the probability that this flower is an `Iris-setosa` flower can be calculated as $P(y = 0|\boldsymbol{x}; \boldsymbol{w}) = 1 - P(y = 1|\boldsymbol{x}; \boldsymbol{w}) = 0.2$ or 20 percent. The predicted probability can then simply be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding plot of the sigmoid function, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

In fact, there are many applications where we are not only interested in the predicted class labels, but where the estimation of the class-membership probability is particularly useful (the output of the sigmoid function prior to applying the threshold function). Logistic regression is used in weather forecasting, for example, not only to predict whether it will rain on a particular day but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in the field of medicine.

Learning the weights of the logistic cost function

You have learned how we can use the logistic regression model to predict probabilities and class labels; now, let's briefly talk about how we fit the parameters of the model, for instance the weights, $\boldsymbol{w}$. In the previous chapter, we defined the sum-squared-error cost function as follows:

$$J(\boldsymbol{w}) = \sum_{i} \frac{1}{2} \left(\phi\left(z^{(i)}\right) - y^{(i)}\right)^2$$

We minimized this function in order to learn the weights, $\boldsymbol{w}$, for our Adaline classification model. To explain how we can derive the cost function for logistic regression, let's first define the likelihood, L, that we want to maximize when we build a logistic regression model, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood** function:

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right]$$

Firstly, applying the log function reduces the potential for numerical underflow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.

Now, we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function, *J*, that can be minimized using gradient descent as in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*:

$$\mathrm{J}(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

To get a better grasp of this cost function, let's take a look at the cost that we calculate for one single training example:

$$J(\phi(z), y; \boldsymbol{w}) = -y \log(\phi(z)) - (1 - y)\log(1 - \phi(z))$$

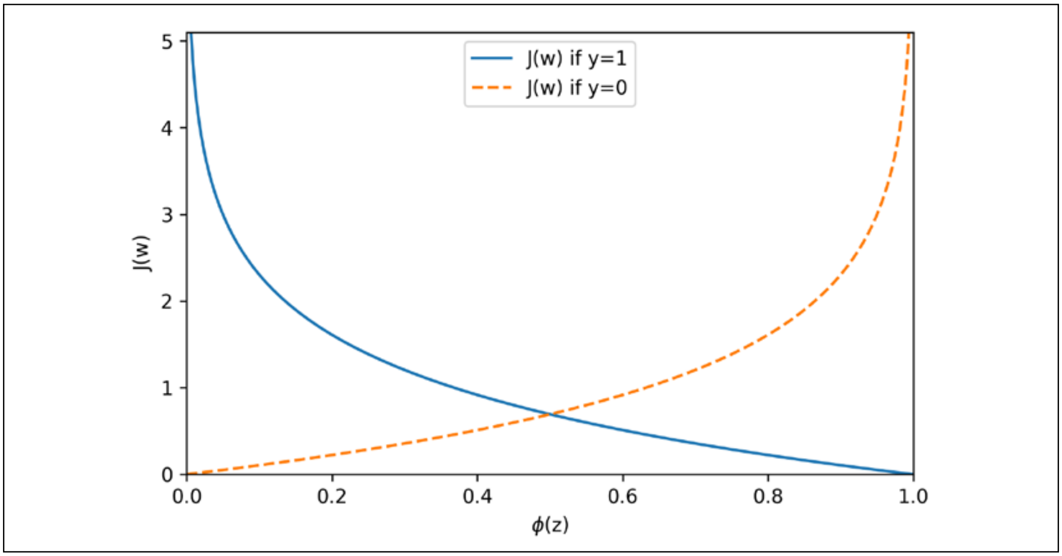
Looking at the equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$:

$$J(\phi(z), y; \boldsymbol{w}) = \begin{cases} -\log\left(\phi(z)\right) & \text{if } y = 1 \\ -\log\left(1 - \phi(z)\right) & \text{if } y = 0 \end{cases}$$

Let's write a short code snippet to create a plot that illustrates the cost of classifying a single training example for different values of $\phi(z)$:

```
>>> def cost_1(z):
...     return - np.log(sigmoid(z))
>>> def cost_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> phi_z = sigmoid(z)
>>> c1 = [cost_1(x) for x in z]
>>> plt.plot(phi_z, c1, label='J(w) if y=1')
>>> c0 = [cost_0(x) for x in z]
>>> plt.plot(phi_z, c0, linestyle='--', label='J(w) if y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\phi$(z)')
>>> plt.ylabel('J(w)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting plot shows the sigmoid activation on the x-axis in the range 0 to 1 (the inputs to the sigmoid function were z values in the range –10 to 10) and the associated logistic cost on the y-axis:



We can see that the cost approaches 0 (continuous line) if we correctly predict that an example belongs to class 1. Similarly, we can see on the y-axis that the cost also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the cost goes toward infinity. The main point is that we penalize wrong predictions with an increasingly larger cost.

Converting an Adaline implementation into an algorithm for logistic regression

If we were to implement logistic regression ourselves, we could simply substitute the cost function, J, in our Adaline implementation from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, with the new cost function:

$$J(\boldsymbol{w}) = -\sum_{i} y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)$$

We use this to compute the cost of classifying all training examples per epoch. Also, we need to swap the linear activation function with the sigmoid activation and change the threshold function to return class labels 0 and 1, instead of –1 and 1. If we make those changes to the Adaline code, we will end up with a working logistic regression implementation, as shown here:

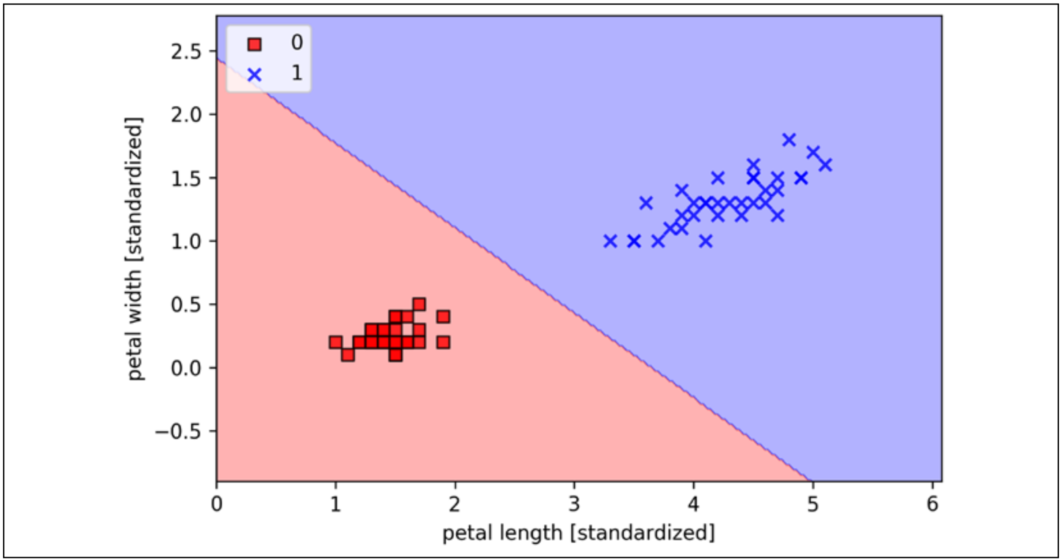
```
class LogisticRegressionGD(object):
    """Logistic Regression Classifier using gradient descent.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight
        initialization.


    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Logistic cost function value in each epoch.

    """
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples is the number of
            examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.
        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()

            # note that we compute the logistic 'cost' now
            # instead of the sum of squared errors cost
            cost = (-y.dot(np.log(output)) -
                        ((1 - y).dot(np.log(1 - output))))
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
        # equivalent to:
        # return np.where(self.activation(self.net_input(X))
        #                 >= 0.5, 1, 0)
```

When we fit a logistic regression model, we have to keep in mind that it only works for binary classification tasks.

So, let's consider only `Iris-setosa` and `Iris-versicolor` flowers (classes `0` and `1`) and check that our implementation of logistic regression works:

```
>>> X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.05,
...                             n_iter=1000,
...                             random_state=1)
>>> lrgd.fit(X_train_01_subset,
...          y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                       y=y_train_01_subset,
...                       classifier=lrgd)
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting decision region plot looks as follows:



The gradient descent learning algorithm for logistic regression

Using calculus, we can show that the weight update in logistic regression via gradient descent is equal to the equation that we used in Adaline in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. However, please note that the following derivation of the gradient descent learning rule is intended for readers who are interested in the mathematical concepts behind the gradient descent learning rule for logistic regression. It is not essential for following the rest of this chapter.

Let's start by calculating the partial derivative of the log-likelihood function with respect to the *j*th weight:

$$\frac{\partial}{\partial w_j} l(\boldsymbol{w}) = \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z)$$

Before we continue, let's also calculate the partial derivative of the sigmoid function:

$$\frac{\partial}{\partial z}\phi(z) = \frac{\partial}{\partial z}\frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2}\,e^{-z} = \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) = \phi(z)\big(1-\phi(z)\big)$$

Now, we can resubstitute $\frac{\partial}{\partial z}\phi(z) = \phi(z)\big(1-\phi(z)\big)$ in our first equation to obtain the following:

$$\begin{aligned}
\left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\frac{\partial}{\partial w_j}\phi(z) &= \left(y\frac{1}{\phi(z)} - (1-y)\frac{1}{1-\phi(z)}\right)\phi(z)\big(1-\phi(z)\big)\frac{\partial}{\partial w_j}z \\
&= \Big(y\big(1-\phi(z)\big) - (1-y)\phi(z)\Big)x_j \\
&= \big(y-\phi(z)\big)x_j
\end{aligned}$$

Remember that the goal is to find the weights that maximize the log-likelihood so that we perform the update for each weight as follows:

$$w_j \coloneqq w_j + \eta\sum_{i=1}^{n}\left(y^{(i)} - \phi\big(z^{(i)}\big)\right)x_j{}^{(i)}$$

Since we update all weights simultaneously, we can write the general update rule as follows:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} + \Delta\boldsymbol{w}$$

We define $\Delta\boldsymbol{w}$ as follows:

$$\Delta\boldsymbol{w} = \eta\nabla l(\boldsymbol{w})$$

Since maximizing the log-likelihood is equal to minimizing the cost function, *J*, that we defined earlier, we can write the gradient descent update rule as follows:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_{i=1}^{n}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)x_j{}^{(i)}$$

$$\boldsymbol{w} \coloneqq \boldsymbol{w} + \Delta\boldsymbol{w}, \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$$

This is equal to the gradient descent rule for Adaline in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*.

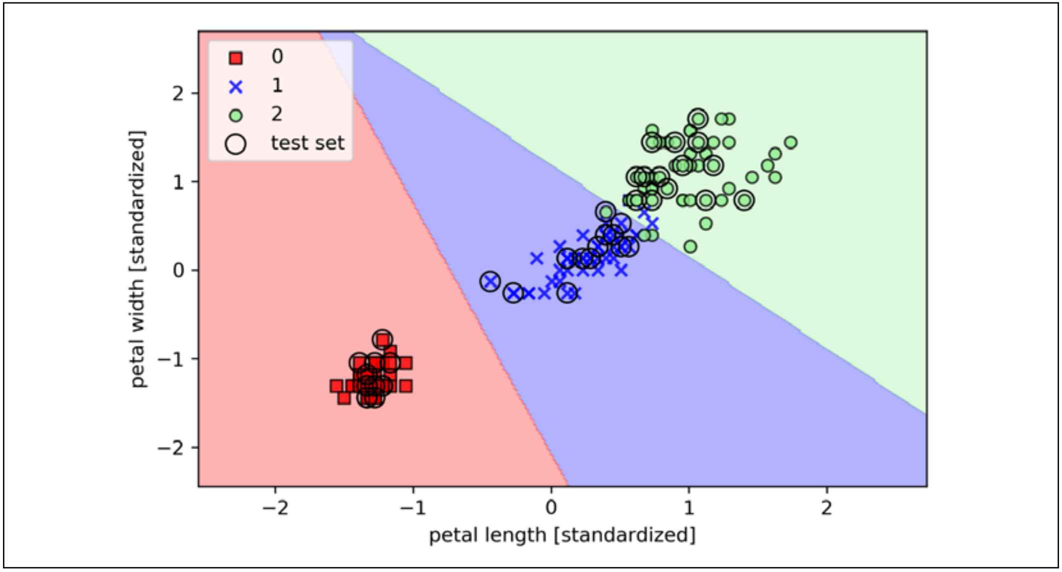
Training a logistic regression model with scikit-learn

We just went through useful coding and math exercises in the previous subsection, which helped to illustrate the conceptual differences between Adaline and logistic regression. Now, let's learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off the shelf. Note that in recent versions of scikit-learn, the technique used for multiclass classification, multinomial, or OvR, is chosen automatically. In the following code example, we will use the `sklearn.linear_model.LogisticRegression` class as well as the familiar `fit` method to train the model on all three classes in the standardized flower training dataset. Also, we set `multi_class='ovr'` for illustration purposes. As an exercise for the reader, you may want to compare the results with `multi_class='multinomial'`. Note that the `multinomial` setting is usually recommended in practice for mutually exclusive classes, such as those found in the Iris dataset. Here, "mutually exclusive" means that each training example can only belong to a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Now, let's have a look at the code example:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, random_state=1,
...                         solver='lbfgs', multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=lr,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After fitting the model on the training data, we plotted the decision regions, training examples, and test examples, as shown in the following plot:



Note that there exist many different optimization algorithms for solving optimization problems. For minimizing convex loss functions, such as the logistic regression loss, it is recommended to use more advanced approaches than regular stochastic gradient descent (SGD). In fact, scikit-learn implements a whole range of such optimization algorithms, which can be specified via the solver parameter, namely, `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, and `'saga'`.

While the logistic regression loss is convex, most optimization algorithms should converge to the global loss minimum with ease. However, there are certain advantages of using one algorithm over the other. For instance, in the current version (v 0.21), scikit-learn uses `'liblinear'` as a default, which cannot handle the multinomial loss and is limited to the OvR scheme for multi-class classification. However, in future versions of scikit-learn (that is, v 0.22), the default solver will be changed to `'lbfgs'`, which stands for the **limited-memory Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) algorithm (`https://en.wikipedia.org/wiki/Limited-memory_BFGS`) and is more flexible in this regard. To adopt this new default choice, we will specify `solver='lbfgs'` explicitly when using logistic regression throughout this book.

Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter C?" We will discuss this parameter in the next subsection, where we will introduce the concepts of overfitting and regularization. However, before we move on to those topics, let's finish our discussion of class-membership probabilities.

The probability that training examples belong to a certain class can be computed using the `predict_proba` method. For example, we can predict the probabilities of the first three examples in the test dataset as follows:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

This code snippet returns the following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

The first row corresponds to the class-membership probabilities of the first flower, the second row corresponds to the class-membership probabilities of the second flower, and so forth. Notice that the columns all sum up to one, as expected. (You can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`.)

The highest value in the first row is approximately 0.85, which means that the first example belongs to class three (`Iris-virginica`) with a predicted probability of 85 percent. So, as you may have already noticed, we can get the predicted class labels by identifying the largest column in each row, for example, using NumPy's `argmax` function:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

The returned class indices are shown here (they correspond to `Iris-virginica`, `Iris-setosa`, and `Iris-setosa`):

```
array([2, 0, 0])
```

In the preceding code example, we computed the conditional probabilities and converted these into class labels manually by using NumPy's `argmax` function. In practice, the more convenient way of obtaining class labels when using scikit-learn is to call the `predict` method directly:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

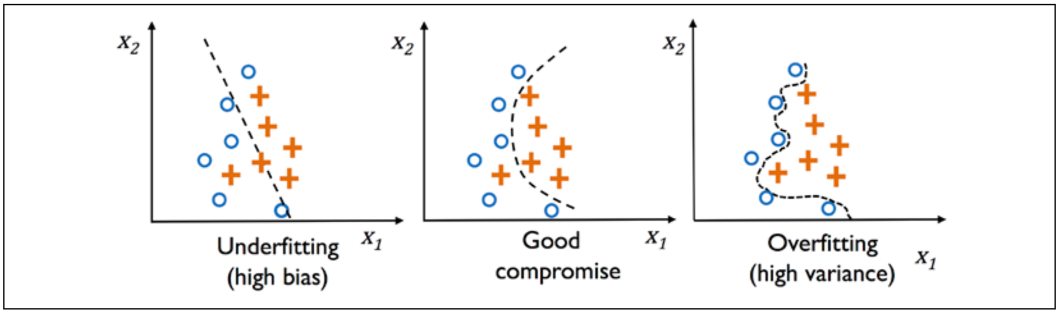
Lastly, a word of caution if you want to predict the class label of a single flower example: scikit-learn expects a two-dimensional array as data input; thus, we have to convert a single row slice into such a format first. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as demonstrated here:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

Although we have only encountered linear models for classification so far, the problems of overfitting and underfitting can be best illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries, as shown in the following figure:



The bias-variance tradeoff

Often, researchers use the terms "bias" and "variance" or "bias-variance tradeoff" to describe the performance of a model—that is, you may stumble upon talks, books, or articles where people say that a model has a "high variance" or "high bias." So, what does that mean? In general, we might say that "high variance" is proportional to overfitting and "high bias" is proportional to underfitting.

In the context of machine learning models, **variance** measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, **bias** measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

If you are interested in the technical specification and derivation of the "bias" and "variance" terms, I've written about it in my lecture notes here: `https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/08_eval-intro_notes.pdf`.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.

The concept behind regularization is to introduce additional information (bias) to penalize extreme parameter (weight) values. The most common form of regularization is so-called **L2 regularization** (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called **regularization parameter**.

Regularization and feature normalization

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

The cost function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

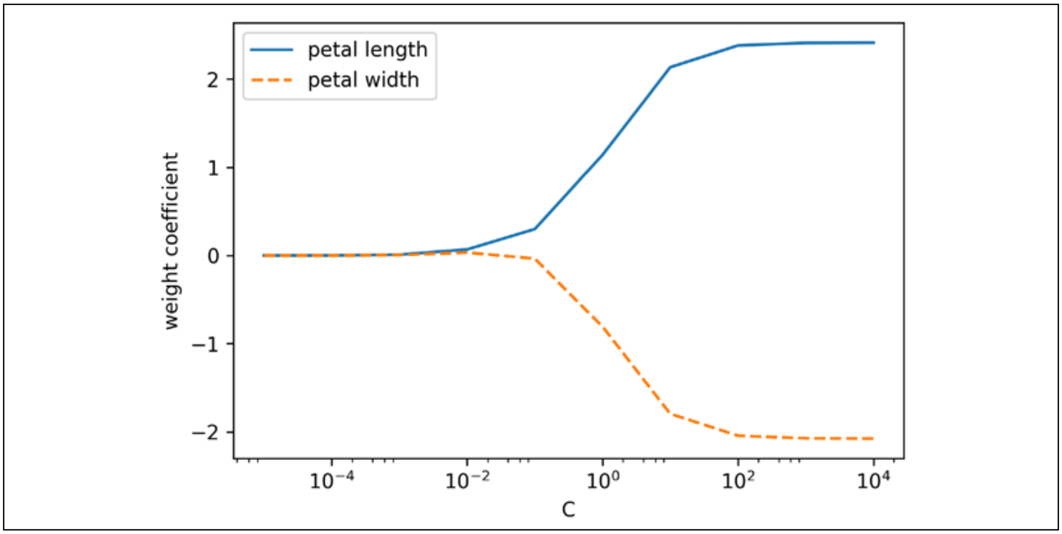
Via the regularization parameter, λ, we can then control how well we fit the training data, while keeping the weights small. By increasing the value of λ, we increase the regularization strength.

The parameter, `C`, that is implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. The term `C` is directly related to the regularization parameter, λ, which is its inverse. Consequently, decreasing the value of the inverse regularization parameter, `C`, means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c, random_state=1,
...                             solver='lbfgs', multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='petal width')
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing the preceding code, we fitted 10 logistic regression models with different values for the inverse-regularization parameter, `C`. For the purposes of illustration, we only collected the weight coefficients of class `1` (here, the second class in the dataset: `Iris-versicolor`) versus all classifiers—remember that we are using the OvR technique for multiclass classification.

As we can see in the resulting plot, the weight coefficients shrink if we decrease parameter `C`, that is, if we increase the regularization strength:

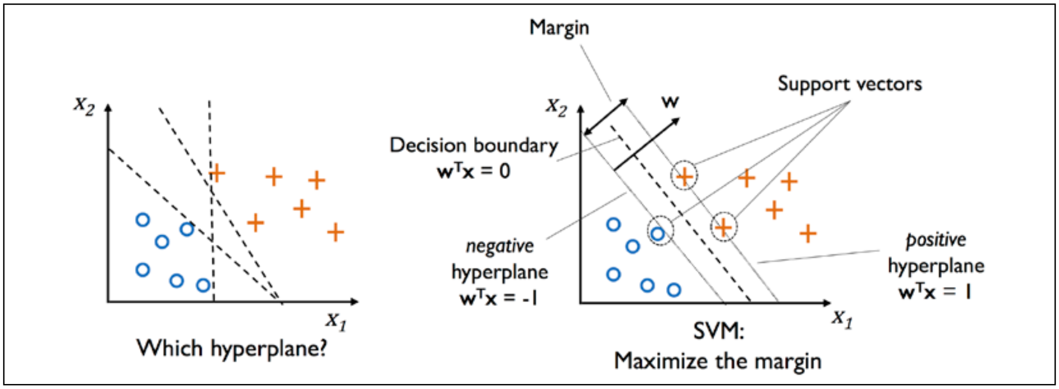


An additional resource on logistic regression

Since an in-depth coverage of the individual classification algorithms exceeds the scope of this book, *Logistic Regression: From Introductory to Advanced Concepts and Applications, Dr. Scott Menard, Sage Publications, 2009*, is recommended to readers who want to learn more about logistic regression.

Maximum margin classification with support vector machines

Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called **support vectors**. This is illustrated in the following figure:



Maximum margin intuition

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting. To get an idea of the margin maximization, let's take a closer look at those positive and negative hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{pos} = 1 \quad (1)$$

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{neg} = -1 \quad (2)$$

If we subtract those two linear equations (1) and (2) from each other, we get:

$$\Rightarrow \boldsymbol{w}^T(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg}) = 2$$

We can normalize this equation by the length of the vector ***w***, which is defined as follows:

$$\|\boldsymbol{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

So, we arrive at the following equation:

$$\frac{\boldsymbol{w}^{\boldsymbol{T}}(\boldsymbol{x}_{pos} - \boldsymbol{x}_{neg})}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}$$

The left side of the preceding equation can then be interpreted as the distance between the positive and negative hyperplane, which is the so-called **margin** that we want to maximize.

Now, the objective function of the SVM becomes the maximization of this margin by maximizing $\frac{2}{\|\boldsymbol{w}\|}$ under the constraint that the examples are classified correctly, which can be written as:

$$w_{\boldsymbol{0}} + \boldsymbol{w}^T\boldsymbol{x}^{(i)} \geq 1 \quad if\ y^{(i)} = 1$$

$$w_{\boldsymbol{0}} + \boldsymbol{w}^T\boldsymbol{x}^{(i)} \leq -1 \quad if\ y^{(i)} = -1$$

$$\text{for } i\ = 1 \ldots N$$

Here, *N* is the number of examples in our dataset.

These two equations basically say that all negative-class examples should fall on one side of the negative hyperplane, whereas all the positive-class examples should fall behind the positive hyperplane, which can also be written more compactly as follows:

$$y^{(i)}(w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)}) \geq 1 \qquad \forall_i$$

In practice, though, it is easier to minimize the reciprocal term, $\frac{1}{2}\|\boldsymbol{w}\|^2$, which can be solved by quadratic programming. However, a detailed discussion about quadratic programming is beyond the scope of this book. You can learn more about SVMs in *The Nature of Statistical Learning Theory*, *Springer Science+Business Media*, *Vladimir Vapnik*, 2000, or read Chris J.C. Burges' excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (*Data Mining and Knowledge Discovery*, *2(2)*: 121-167, *1998*).

Dealing with a nonlinearly separable case using slack variables

Although we don't want to dive much deeper into the more involved mathematical concepts behind the maximum-margin classification, let's briefly mention the slack variable, ξ, which was introduced by Vladimir Vapnik in 1995 and led to the so-called **soft-margin classification**. The motivation for introducing the slack variable was that the linear constraints need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate cost penalization.

The positive-valued slack variable is simply added to the linear constraints:

$$w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)} \quad if\ y^{(i)} = 1$$

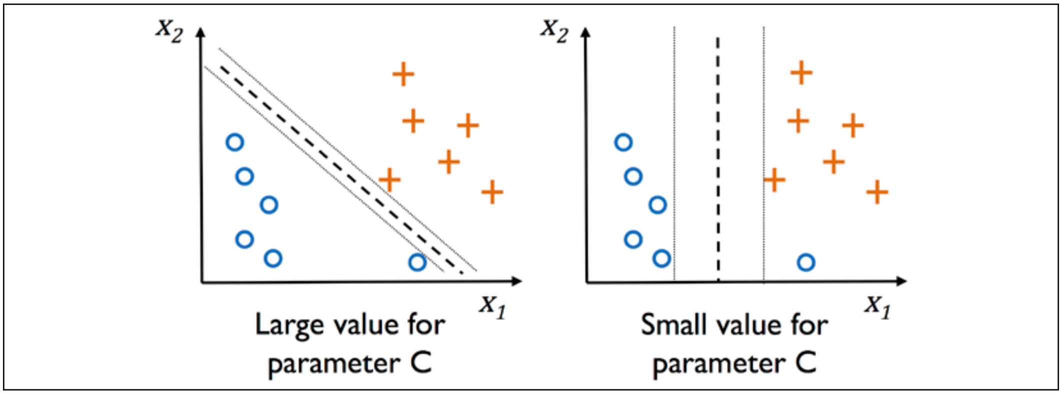
$$w_{\mathbf{0}} + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)} \quad if\ y^{(i)} = -1$$

$$\text{for } i\ = 1 \ldots N$$

Here, N is the number of examples in our dataset. So, the new objective to be minimized (subject to the constraints) becomes:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_i \xi^{(i)}\right)$$

Via the variable, `C`, we can then control the penalty for misclassification. Large values of `C` correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for `C`. We can then use the `C` parameter to control the width of the margin and therefore tune the bias-variance tradeoff, as illustrated in the following figure:

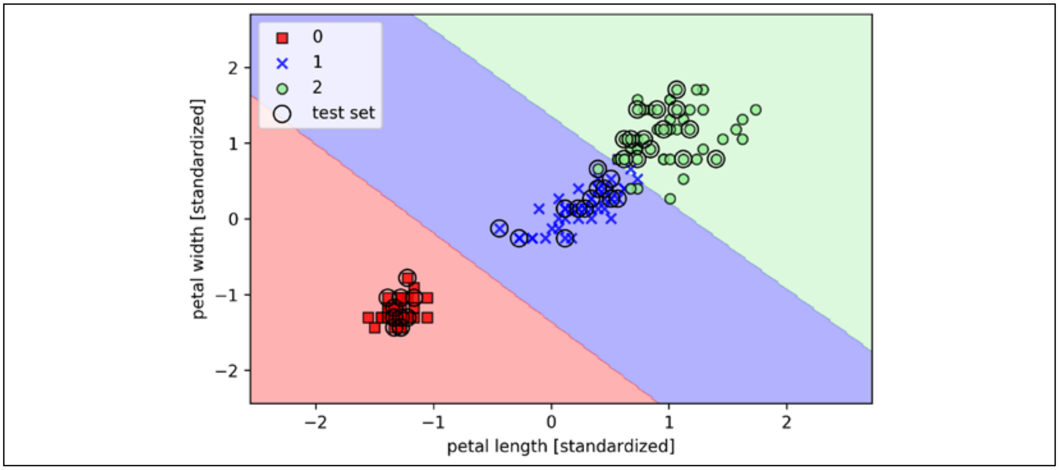


This concept is related to regularization, which we discussed in the previous section in the context of regularized regression, where decreasing the value of `C` increases the bias and lowers the variance of the model.

Now that we have learned the basic concepts behind a linear SVM, let's train an SVM model to classify the different flowers in our Iris dataset:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=svm,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The three decision regions of the SVM, visualized after training the classifier on the Iris dataset by executing the preceding code example, are shown in the following plot:



Logistic regression versus SVMs

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage that it is a simpler model and can be implemented more easily. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.

Alternative implementations in scikit-learn

The scikit-learn library's `LogisticRegression` class, which we used in the previous sections, makes use of the LIBLINEAR library, which is a highly optimized C/C++ library developed at the National Taiwan University (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`).

Similarly, the `SVC` class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`).

The advantage of using LIBLINEAR and LIBSVM over native Python implementations is that they allow the extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the `SGDClassifier` class, which also supports online learning via the `partial_fit` method. The concept behind the `SGDClassifier` class is similar to the stochastic gradient algorithm that we implemented in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, for Adaline. We could initialize the SGD version of the perceptron, logistic regression, and an SVM with default parameters, as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

Solving nonlinear problems using a kernel SVM

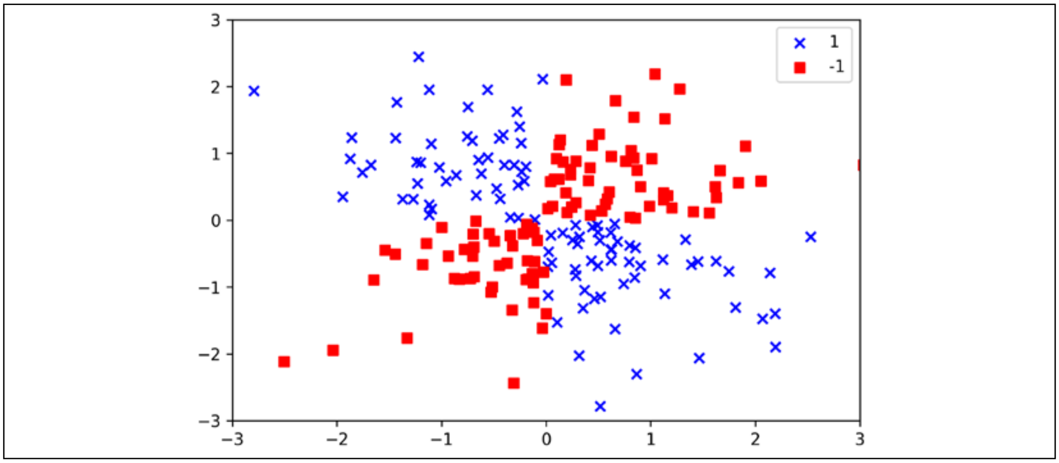
Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily **kernelized** to solve nonlinear classification problems. Before we discuss the main concept behind the so-called **kernel SVM**, the most common variant of SVMs, let's first create a synthetic dataset to see what such a nonlinear classification problem may look like.

Kernel methods for linearly inseparable data

Using the following code, we will create a simple dataset that has the form of an XOR gate using the `logical_or` function from NumPy, where 100 examples will be assigned the class label `1`, and 100 examples will be assigned the class label `-1`:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(1)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
...                        X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)
>>> plt.scatter(X_xor[y_xor == 1, 0],
...                X_xor[y_xor == 1, 1],
...                c='b', marker='x',
...                label='1')
>>> plt.scatter(X_xor[y_xor == -1, 0],
...                X_xor[y_xor == -1, 1],
...                c='r',
...                marker='s',
...                label='-1')
>>> plt.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we will have an XOR dataset with random noise, as shown in the following plot:

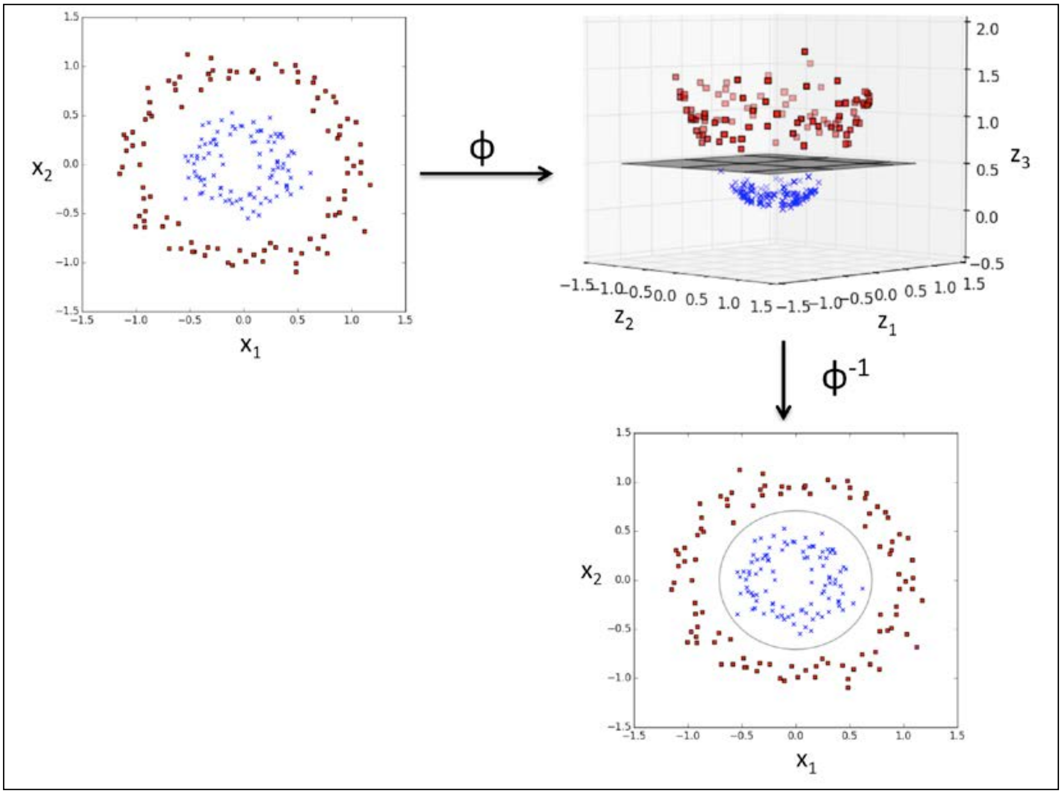


Obviously, we would not be able to separate the examples from the positive and negative class very well using a linear hyperplane as a decision boundary via the linear logistic regression or linear SVM model that we discussed in earlier sections.

The basic idea behind **kernel methods** to deal with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, ϕ, where the data becomes linearly separable. As shown in the following plot, we can transform a two-dimensional dataset into a new three-dimensional feature space, where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

This allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space:



Using the kernel trick to find separating hyperplanes in a high-dimensional space

To solve a nonlinear problem using an SVM, we would transform the training data into a higher-dimensional feature space via a mapping function, ϕ, and train a linear SVM model to classify the data in this new feature space. Then, we could use the same mapping function, ϕ, to transform new, unseen data to classify it using the linear SVM model.

However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called **kernel trick** comes into play.

Although we did not go into much detail about how to solve the quadratic programming task to train an SVM, in practice, we just need to replace the dot product $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ by $\phi\left(\boldsymbol{x}^{(i)}\right)^T\phi\left(\boldsymbol{x}^{(j)}\right)$. In order to save the expensive step of calculating this dot product between two points explicitly, we define a so-called **kernel function**:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

One of the most widely used kernels is the **radial basis function** (**RBF**) kernel, which can simply be called the **Gaussian kernel**:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

This is often simplified to:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)}-\boldsymbol{x}^{(j)}\right\|^2\right)$$

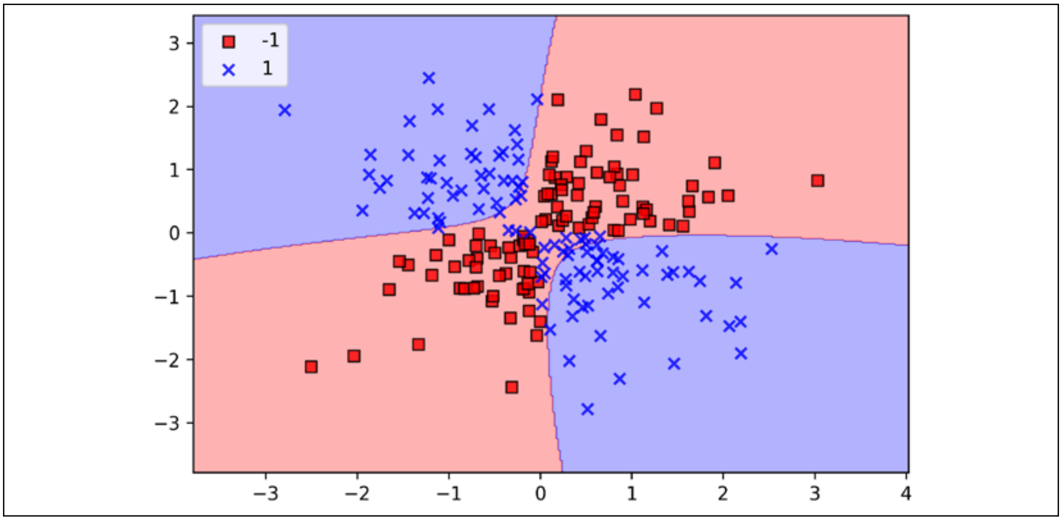
Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter to be optimized.

Roughly speaking, the term "kernel" can be interpreted as a **similarity function** between a pair of examples. The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples).

Now that we have covered the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the `SVC` class from scikit-learn that we imported earlier and replace the `kernel='linear'` parameter with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

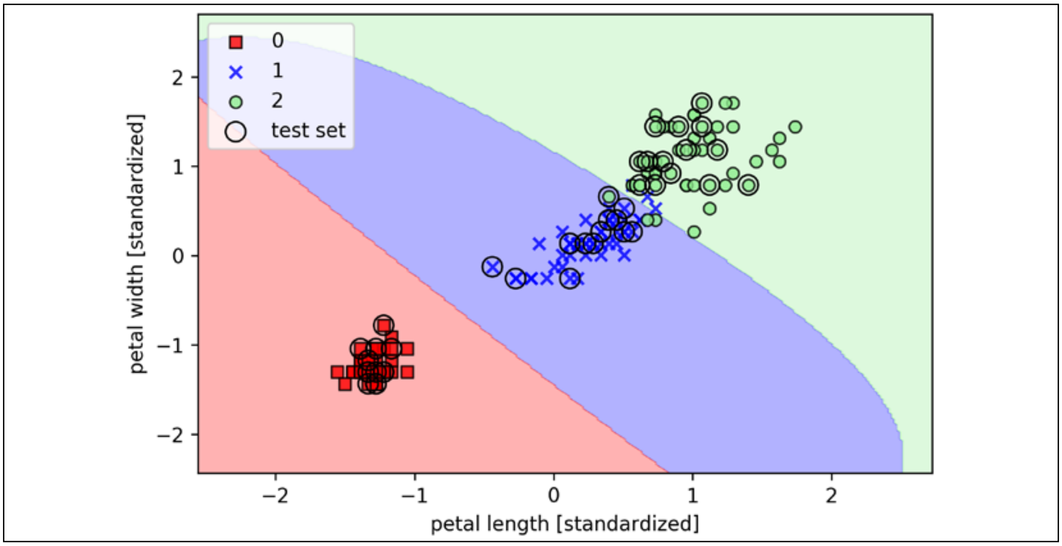
As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:



The γ parameter, which we set to `gamma=0.1`, can be understood as a cut-off parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training examples, which leads to a tighter and bumpier decision boundary. To get a better understanding of γ, let's apply an RBF kernel SVM to our Iris flower dataset:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

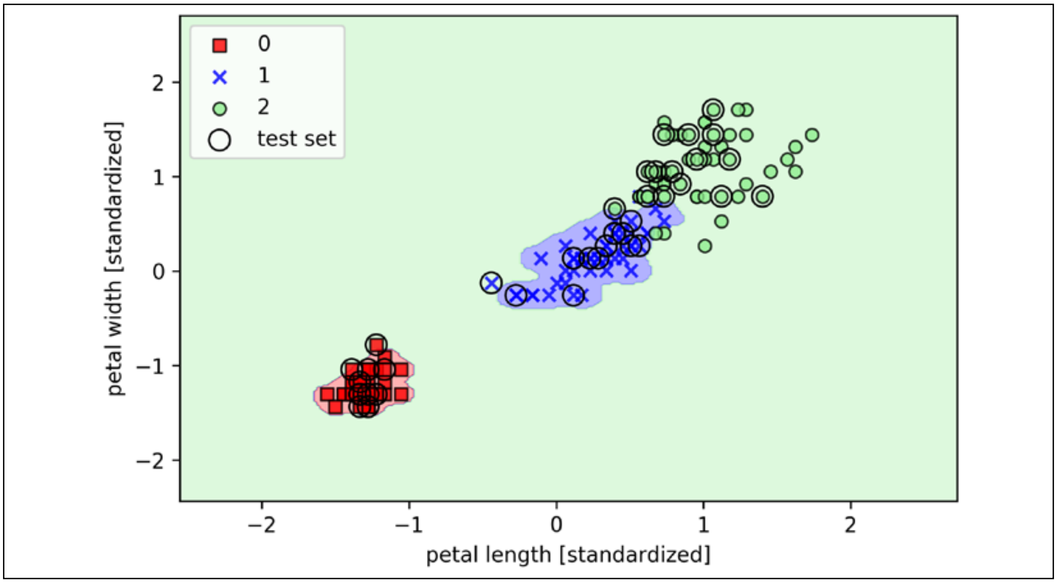
Since we chose a relatively small value for γ, the resulting decision boundary of the RBF kernel SVM model will be relatively soft, as shown in the following plot:



Now, let's increase the value of γ and observe the effect on the decision boundary:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In the resulting plot, we can now see that the decision boundary around the classes `0` and `1` is much tighter using a relatively large value of γ:

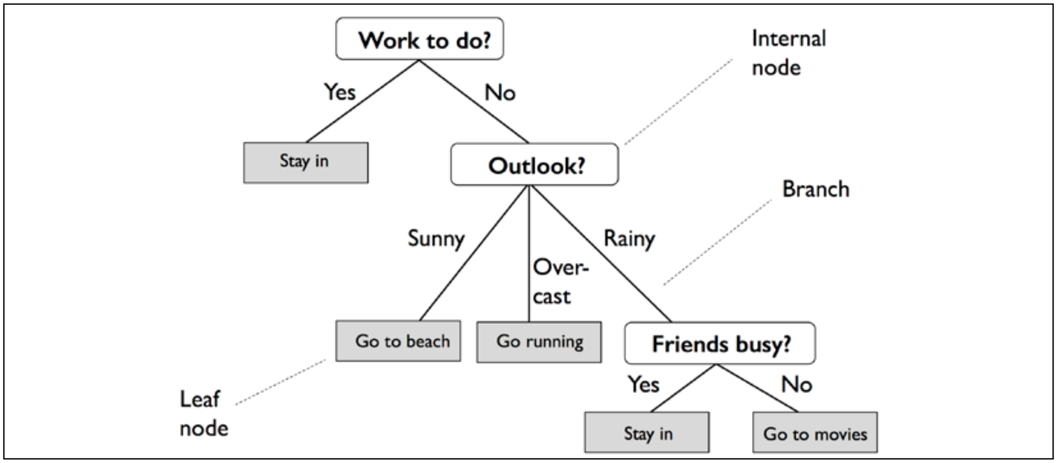


Although the model fits the training dataset very well, such a classifier will likely have a high generalization error on unseen data. This illustrates that the γ parameter also plays an important role in controlling overfitting or variance when the algorithm is too sensitive to fluctuations in the training dataset.

Decision tree learning

Decision tree classifiers are attractive models if we care about interpretability. As the name "decision tree" suggests, we can think of this model as breaking down our data by making a decision based on asking a series of questions.

Let's consider the following example in which we use a decision tree to decide upon an activity on a particular day:



Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples. Although the preceding figure illustrates the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers, like in the Iris dataset. For example, we could simply define a cut-off value along the **sepal width** feature axis and ask a binary question: "Is the sepal width ≥ 2.8 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which will be explained in more detail in the following section. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the training examples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to **prune** the tree by setting a limit for the maximal depth of the tree.

Maximizing IG – getting the most bang for your buck

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here, *f* is the feature to perform the split; D_p and D_j are the dataset of the parent and *j*th child node; *I* is our **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the *j*th child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all **non-empty** classes ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \, log_2 \, p(i|t)$$

Here, $p(i|t)$ is the proportion of the examples that belong to class *i* for a particular node, *t*. The entropy is therefore 0 if all examples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i = 1|t) = 1$ or $p(i = 0|t) = 0$. If the classes are distributed uniformly with $p(i = 1|t) = 0.5$ and $p(i = 0|t) = 0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.

The Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting (*c* = 2):

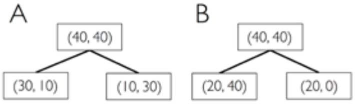
$$I_G(t) = 1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

However, in practice, both Gini impurity and entropy typically yield very similar results, and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs.

Another impurity measure is the classification error:

$$I_E(t) = 1 - \max\{\, p(i|t)\}$$

This is a useful criterion for pruning but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes. We can illustrate this by looking at the two possible splitting scenarios shown in the following figure:



We start with a dataset, D_p, at the parent node, which consists of 40 examples from class 1 and 40 examples from class 2 that we split into two datasets, D_{left} and D_{right}. The information gain using the classification error as a splitting criterion would be the same ($IG_E = 0.25$) in both scenarios, A and B:

$$I_E(D_p) = 1 - 0.5 = 0.5$$

$$A\colon\quad I_E(D_{left}) = 1 - \frac{3}{4} = 0.25$$

$$A\colon\quad I_E(D_{Right}) = 1 - \frac{3}{4} = 0.25$$

$$A\colon\quad IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B\colon\quad I_E(D_{left}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B\colon\quad I_E(D_{Right}) = 1 - 1 = 0$$

$$B\colon\quad IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

However, the Gini impurity would favor the split in scenario B ($IG_G = 0.1\bar{\bar{6}}$) over scenario A ($IG_G = 0.125$), which is indeed purer:

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A: \quad I_G(D_{left}) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad I_G(D_{right}) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

$$B: \quad I_G(D_{left}) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\overline{4}$$

$$B: \quad I_G(D_{right}) = 1 - (1^2 + 0^2) = 0$$

$$B: \quad IG_G = 0.5 - \frac{6}{8}0.\overline{4} - \ 0 = 0.\overline{16}$$

Similarly, the entropy criterion would also favor scenario B ($IG_H = 0.31$) over scenario A ($IG_H = 0.19$):

$$I_H(D_p) = -\big(0.5\ log_2(0.5) + 0.5\ log_2(0.5)\big) = 1$$

$$A: I_H(D_{left}) = -\left(\frac{3}{4}\ log_2\left(\frac{3}{4}\right) + \frac{1}{4}\ log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: I_H(D_{right}) = -\left(\frac{1}{4}\ log_2\left(\frac{1}{4}\right) + \frac{3}{4}\ log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: I_H(D_{left}) = -\left(\frac{2}{6}\ log_2\left(\frac{2}{6}\right) + \frac{4}{6}\ log_2\left(\frac{4}{6}\right)\right) = 0.92$$

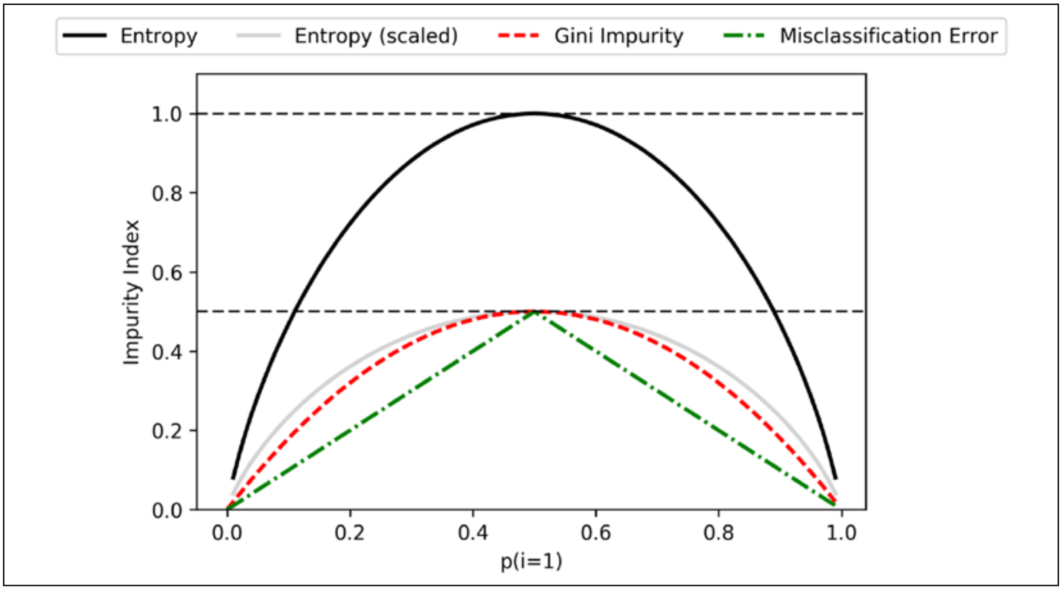
$$B: I_H(D_{right}) = 0$$

$$B: IG_H = 1 - \frac{6}{8}0.92 - \ 0 = 0.31$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range [0, 1] for class 1. Note that we will also add a scaled version of the entropy (entropy / 2) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                          ['Entropy', 'Entropy (scaled)',
...                           'Gini impurity',
...                           'Misclassification error'],
...                          ['-', '-', '--', '-.'],
...                          ['black', 'lightgray',
...                           'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                   linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('impurity index')
```

The plot produced by the preceding code example is as follows:

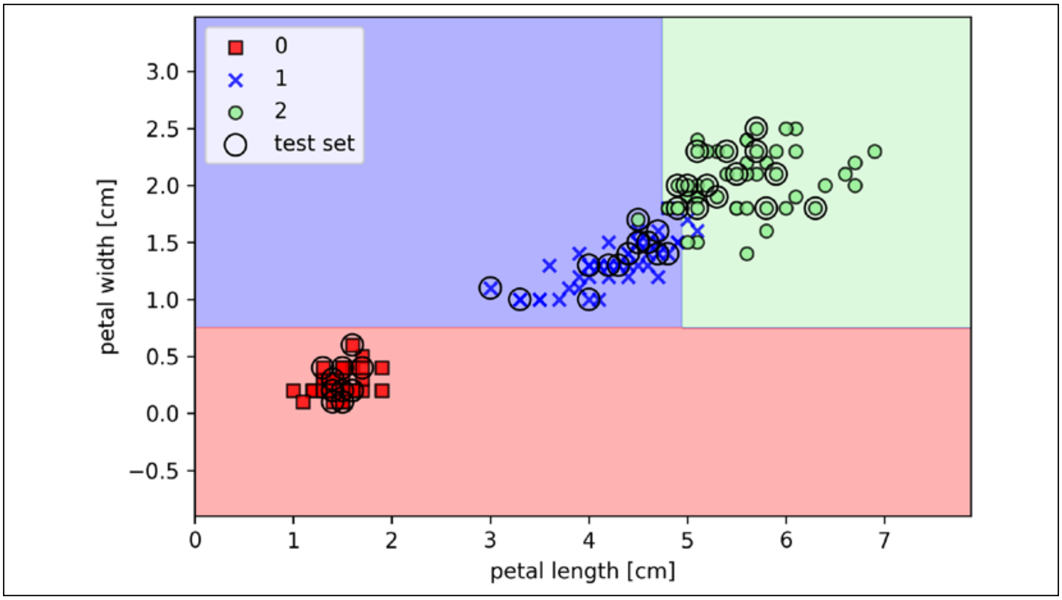


Building a decision tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting. Using scikit-learn, we will now train a decision tree with a maximum depth of 4, using Gini impurity as a criterion for impurity. Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms. The code is as follows:

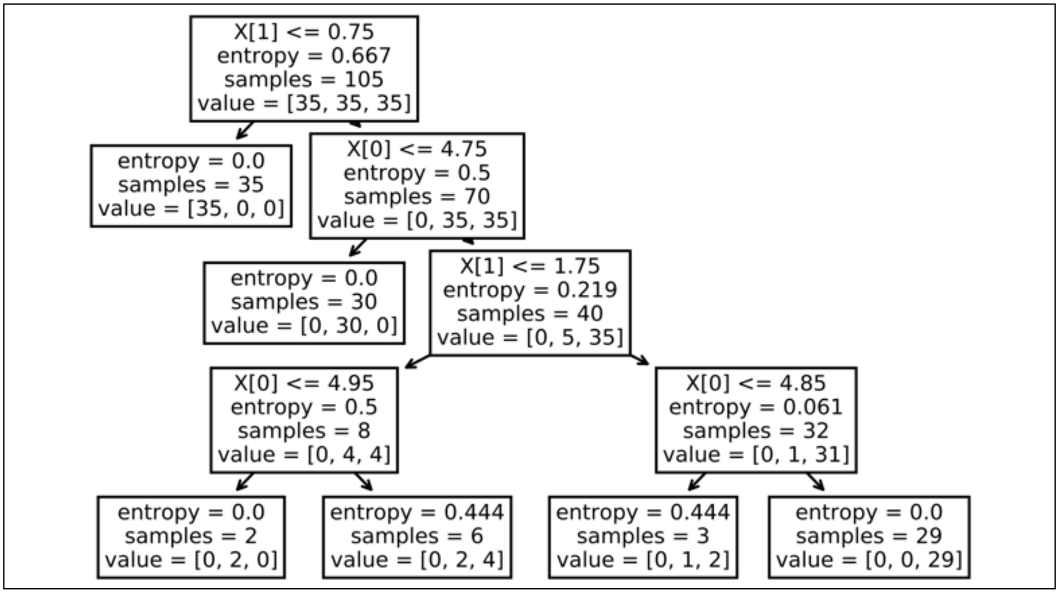
```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree_model = DecisionTreeClassifier(criterion='gini',
...                                     max_depth=4,
...                                     random_state=1)
>>> tree_model.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined,
...                       y_combined,
...                       classifier=tree_model,
...                       test_idx=range(105, 150))
>>> plt.xlabel('petal length [cm]')
>>> plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code example, we get the typical axis-parallel decision boundaries of the decision tree:



A nice feature in scikit-learn is that it allows us to readily visualize the decision tree model after training via the following code:

```
>>> from sklearn import tree
>>> tree.plot_tree(tree_model)
>>> plt.show()
```



However, nicer visualizations can be obtained by using the Graphviz program as a backend for plotting scikit-learn decision trees. This program is freely available from `http://www.graphviz.org` and is supported by Linux, Windows, and macOS. In addition to Graphviz, we will use a Python library called PyDotPlus, which has capabilities similar to Graphviz and allows us to convert `.dot` data files into a decision tree image file. After you have installed Graphviz (by following the instructions on `http://www.graphviz.org/download`), you can install PyDotPlus directly via the pip installer, for example, by executing the following command in your command-line terminal:

```
> pip3 install pydotplus
```

Installing PyDotPlus prerequisites

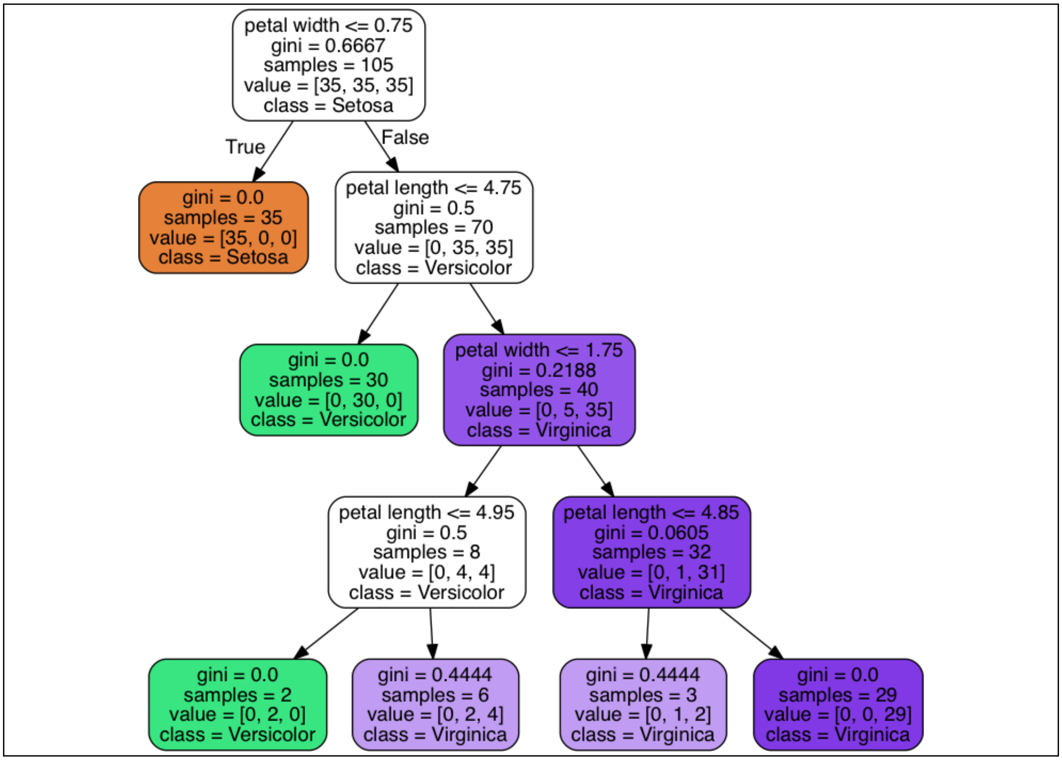
Note that on some systems, you may have to install the PyDotPlus prerequisites manually by executing the following commands:

```
pip3 install graphviz
pip3 install pyparsing
```

The following code will create an image of our decision tree in PNG format in our local directory:

```
>>> from pydotplus import graph_from_dot_data
>>> from sklearn.tree import export_graphviz
>>> dot_data = export_graphviz(tree_model,
...                            filled=True,
...                            rounded=True,
...                            class_names=['Setosa',
...                                         'Versicolor',
...                                         'Virginica'],
...                            feature_names=['petal length',
...                                           'petal width'],
...                            out_file=None)
>>> graph = graph_from_dot_data(dot_data)
>>> graph.write_png('tree.png')
```

By using the `out_file=None` setting, we directly assigned the DOT data to a `dot_data` variable, instead of writing an intermediate `tree.dot` file to disk. The arguments for `filled`, `rounded`, `class_names`, and `feature_names` are optional, but make the resulting image file visually more appealing by adding color, rounding the box edges, showing the name of the majority class label at each node, and displaying the feature name in each splitting criterion. These settings resulted in the following decision tree image:



Looking at the decision tree figure, we can now nicely trace back the splits that the decision tree determined from our training dataset. We started with 105 examples at the root and split them into two child nodes with 35 and 70 examples, using the petal width cut-off ≤ 0.75 cm. After the first split, we can see that the left child node is already pure and only contains examples from the `Iris-setosa` class (Gini impurity = 0). The further splits on the right are then used to separate the examples from the `Iris-versicolor` and `Iris-virginica` class.

Looking at this tree, and the decision region plot of the tree, we can see that the decision tree does a very good job of separating the flower classes. Unfortunately, scikit-learn currently does not implement functionality to manually post-prune a decision tree. However, we could go back to our previous code example, change the `max_depth` of our decision tree to `3`, and compare it to our current model, but we leave this as an exercise for the interested reader.

Combining multiple decision trees via random forests

Ensemble methods have gained huge popularity in applications of machine learning during the last decade due to their good classification performance and robustness toward overfitting. While we are going to cover different ensemble methods, including **bagging** and **boosting**, later in *Chapter 7*, *Combining Different Models for Ensemble Learning*, let's discuss the decision tree-based **random forest** algorithm, which is known for its good scalability and ease of use. A random forest can be considered as an **ensemble** of decision trees. The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting. The random forest algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap** sample of size *n* (randomly choose *n* examples from the training dataset with replacement).
2. Grow a decision tree from the bootstrap sample. At each node:
    a. Randomly select *d* features without replacement.
    b. Split the node using the feature that provides the best split according to the objective function, for instance, maximizing the information gain.
3. Repeat the steps 1-2 *k* times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote**. Majority voting will be discussed in more detail in *Chapter 7*, *Combining Different Models for Ensemble Learning*.

We should note one slight modification in step 2 when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.

Sampling with and without replacement

In case you are not familiar with the terms sampling "with" and "without" replacement, let's walk through a simple thought experiment. let's assume that we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers, 0, 1, 2, 3, and 4, and we draw exactly one number each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.

However, in random sampling with replacement, we always return the drawn number to the urn so that the probability of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling *with* replacement, the samples (numbers) are independent and have a covariance of zero. For example, the results from five rounds of drawing random numbers could look like this:

- Random sampling without replacement: 2, 1, 3, 4, 0
- Random sampling with replacement: 1, 3, 3, 4, 1

Although random forests don't offer the same level of interpretability as decision trees, a big advantage of random forests is that we don't have to worry so much about choosing good hyperparameter values. We typically don't need to prune the random forest since the ensemble model is quite robust to noise from the individual decision trees. The only parameter that we really need to care about in practice is the number of trees, *k*, (step 3) that we choose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

Although it is less common in practice, other hyperparameters of the random forest classifier that can be optimized—using techniques that we will discuss in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*—are the size, *n*, of the bootstrap sample (step 1), and the number of features, *d*, that are randomly chosen for each split (step 2.a), respectively. Via the sample size, *n*, of the bootstrap sample, we control the bias-variance tradeoff of the random forest.

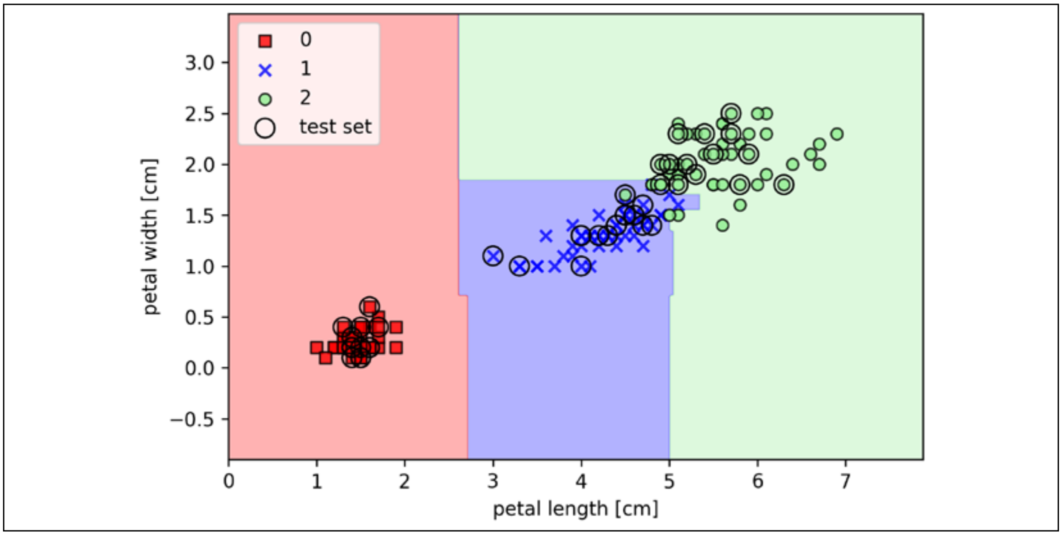
Decreasing the size of the bootstrap sample increases the diversity among the individual trees, since the probability that a particular training example is included in the bootstrap sample is lower. Thus, shrinking the size of the bootstrap samples may increase the *randomness* of the random forest, and it can help to reduce the effect of overfitting. However, smaller bootstrap samples typically result in a lower overall performance of the random forest, and a small gap between training and test performance, but a low test performance overall. Conversely, increasing the size of the bootstrap sample may increase the degree of overfitting. Because the bootstrap samples, and consequently the individual decision trees, become more similar to each other, they learn to fit the original training dataset more closely.

In most implementations, including the `RandomForestClassifier` implementation in scikit-learn, the size of the bootstrap sample is chosen to be equal to the number of training examples in the original training dataset, which usually provides a good bias-variance tradeoff. For the number of features, *d*, at each split, we want to choose a value that is smaller than the total number of features in the training dataset. A reasonable default that is used in scikit-learn and other implementations is $d = \sqrt{m}$, where *m* is the number of features in the training dataset.

Conveniently, we don't have to construct the random forest classifier from individual decision trees by ourselves because there is already an implementation in scikit-learn that we can use:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='gini',
...                                 n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('petal length [cm]')
>>> plt.ylabel('petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the preceding code, we should see the decision regions formed by the ensemble of trees in the random forest, as shown in the following plot:



Using the preceding code, we trained a random forest from 25 decision trees via the `n_estimators` parameter and used the Gini impurity measure as a criterion to split the nodes. Although we are growing a very small random forest from a very small training dataset, we used the `n_jobs` parameter for demonstration purposes, which allows us to parallelize the model training using multiple cores of our computer (here, two cores).

K-nearest neighbors – a lazy learning algorithm

The last supervised learning algorithm that we want to discuss in this chapter is the **k-nearest neighbor** (**KNN**) classifier, which is particularly interesting because it is fundamentally different from the learning algorithms that we have discussed so far.

KNN is a typical example of a **lazy learner**. It is called "lazy" not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Parametric versus nonparametric models

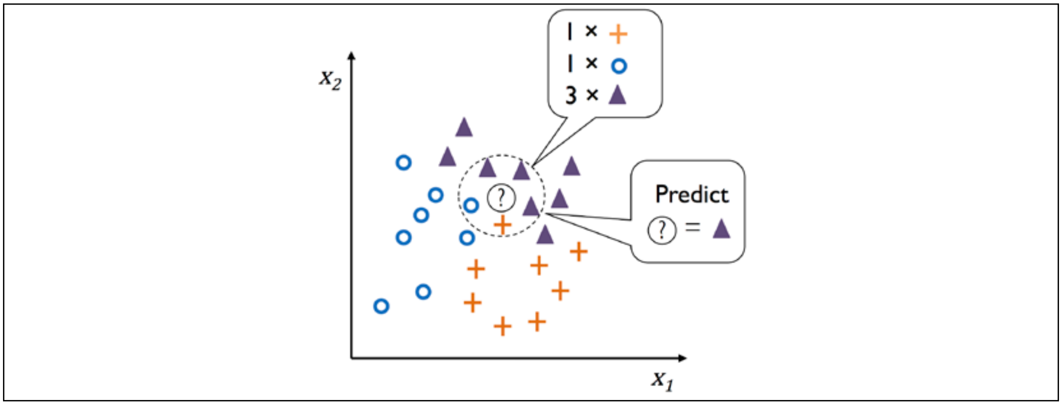
Machine learning algorithms can be grouped into **parametric** and **nonparametric** models. Using parametric models, we estimate parameters from the training dataset to learn a function that can classify new data points without requiring the original training dataset anymore. Typical examples of parametric models are the perceptron, logistic regression, and the linear SVM. In contrast, nonparametric models can't be characterized by a fixed set of parameters, and the number of parameters grows with the training data. Two examples of nonparametric models that we have seen so far are the decision tree classifier/random forest and the kernel SVM.

KNN belongs to a subcategory of nonparametric models that is described as **instance-based learning**. Models based on instance-based learning are characterized by memorizing the training dataset, and lazy learning is a special case of instance-based learning that is associated with no (zero) cost during the learning process.

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of *k* and a distance metric.
2. Find the *k*-nearest neighbors of the data record that we want to classify.
3. Assign the class label by majority vote.

The following figure illustrates how a new data point (**?**) is assigned the triangle class label based on majority voting among its five nearest neighbors.



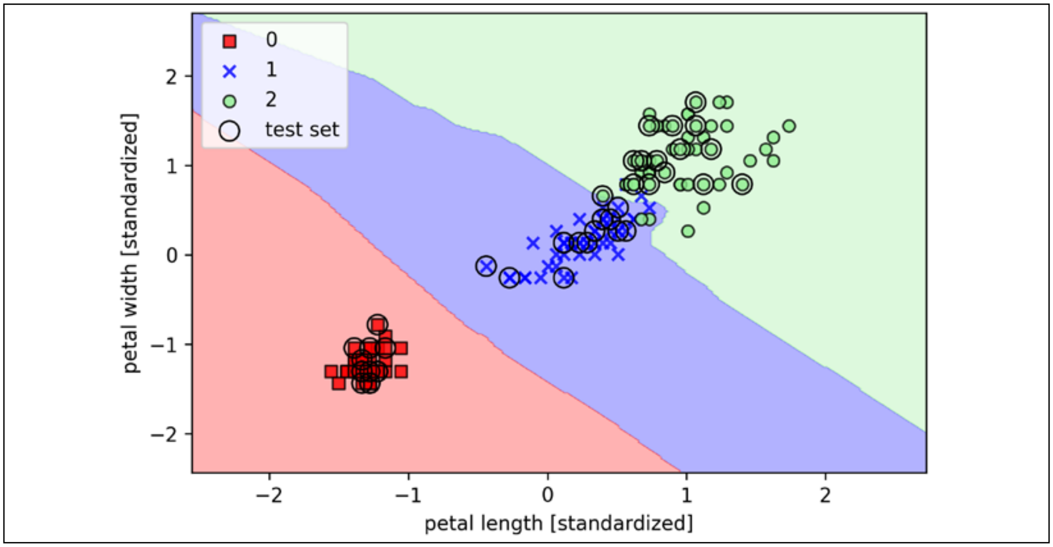
Based on the chosen distance metric, the KNN algorithm finds the *k* examples in the training dataset that are closest (most similar) to the point that we want to classify. The class label of the data point is then determined by a majority vote among its *k* nearest neighbors.

The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data. However, the downside is that the computational complexity for classifying new examples grows linearly with the number of examples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures such as k-d trees (*An Algorithm for Finding Best Matches in Logarithmic Expected Time*, *J. H. Friedman*, *J. L. Bentley*, and *R.A. Finkel*, *ACM transactions on mathematical software* (*TOMS*), *3(3): 209–226, 1977*). Furthermore, we can't discard training examples since no training step is involved. Thus, storage space can become a challenge if we are working with large datasets.

By executing the following code, we will now implement a KNN model in scikit-learn using a Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('petal length [standardized]')
>>> plt.ylabel('petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in the following plot:



Resolving ties

In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the data record to be classified. If the neighbors have similar distances, the algorithm will choose the class label that comes first in the training dataset.

The *right* choice of *k* is crucial to finding a good balance between overfitting and underfitting. We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset. Often, a simple Euclidean distance measure is used for real-value examples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance. The `minkowski` distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance, which can be written as follows:

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_{k} \left| x_k^{(i)} \quad x_k^{(j)} \right|^p},$$

It becomes the Euclidean distance if we set the parameter `p=2` or the Manhattan distance at `p=1`. Many other distance metrics are available in scikit-learn and can be provided to the `metric` parameter. They are listed at `http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html`.

The curse of dimensionality

It is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. We can think of even the closest neighbors as being too far away in a high-dimensional space to give a good estimate.

We discussed the concept of regularization in the section about logistic regression as one way to avoid overfitting. However, in models where regularization is not applicable, such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us to avoid the curse of dimensionality. This will be discussed in more detail in the next chapter.

Summary

In this chapter, you learned about many different machine learning algorithms that are used to tackle linear and nonlinear problems. You have seen that decision trees are particularly attractive if we care about interpretability. Logistic regression is not only a useful model for online learning via SGD, but also allows us to predict the probability of a particular event.

Although SVMs are powerful linear models that can be extended to nonlinear problems via the kernel trick, they have many parameters that have to be tuned in order to make good predictions. In contrast, ensemble methods, such as random forests, don't require much parameter tuning and don't overfit as easily as decision trees, which makes them attractive models for many practical problem domains. The KNN classifier offers an alternative approach to classification via lazy learning that allows us to make predictions without any model training, but with a more computationally expensive prediction step.

However, even more important than the choice of an appropriate learning algorithm is the available data in our training dataset. No algorithm will be able to make good predictions without informative and discriminatory features.

In the next chapter, we will discuss important topics regarding the preprocessing of data, feature selection, and dimensionality reduction, which means that we will need to build powerful machine learning models. Later, in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will see how we can evaluate and compare the performance of our models and learn useful tricks to fine-tune the different algorithms.

4

Building Good Training Datasets – Data Preprocessing

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is absolutely critical to ensure that we examine and preprocess a dataset before we feed it to a learning algorithm. In this chapter, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this chapter are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction

Dealing with missing data

It is not uncommon in real-world applications for our training examples to be missing one or more values for various reasons. There could have been an error in the data collection process, certain measurements may not be applicable, or particular fields could have been simply left blank in a survey, for example. We typically see missing values as blank spaces in our data table or as placeholder strings such as `NaN`, which stands for "not a number," or `NULL` (a commonly used indicator of unknown values in relational databases). Unfortunately, most computational tools are unable to handle such missing values or will produce unpredictable results if we simply ignore them. Therefore, it is crucial that we take care of those missing values before we proceed with further analyses.

In this section, we will work through several practical techniques for dealing with missing values by removing entries from our dataset or imputing missing values from other training examples and features.

Identifying missing values in tabular data

Before we discuss several techniques for dealing with missing values, let's create a simple example data frame from a **comma-separated values** (**CSV**) file to get a better grasp of the problem:

```
>>> import pandas as pd
>>> from io import StringIO

>>> csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
        A        B        C        D
0     1.0      2.0      3.0      4.0
1     5.0      6.0      NaN      8.0
2    10.0     11.0     12.0      NaN
```

Using the preceding code, we read CSV-formatted data into a pandas `DataFrame` via the `read_csv` function and noticed that the two missing cells were replaced by `NaN`. The `StringIO` function in the preceding code example was simply used for the purposes of illustration. It allowed us to read the string assigned to `csv_data` into a pandas `DataFrame` as if it was a regular CSV file on our hard drive.

For a larger `DataFrame`, it can be tedious to look for missing values manually; in this case, we can use the `isnull` method to return a `DataFrame` with Boolean values that indicate whether a cell contains a numeric value (`False`) or if data is missing (`True`). Using the `sum` method, we can then return the number of missing values per column as follows:

```
>>> df.isnull().sum()
A      0
B      0
C      1
```

```
D      1
dtype: int64
```

This way, we can count the number of missing values per column; in the following subsections, we will take a look at different strategies for how to deal with this missing data.

Convenient data handling with pandas' data frames

Although scikit-learn was originally developed for working with NumPy arrays only, it can sometimes be more convenient to preprocess data using pandas' `DataFrame`. Nowadays, most scikit-learn functions support `DataFrame` objects as inputs, but since NumPy array handling is more mature in the scikit-learn API, it is recommended to use NumPy arrays when possible. Note that you can always access the underlying NumPy array of a `DataFrame` via the `values` attribute before you feed it into a scikit-learn estimator:

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

Eliminating training examples or features with missing values

One of the easiest ways to deal with missing data is simply to remove the corresponding features (columns) or training examples (rows) from the dataset entirely; rows with missing values can easily be dropped via the `dropna` method:

```
>>> df.dropna(axis=0)
      A    B    C    D
0   1.0  2.0  3.0  4.0
```

Similarly, we can drop columns that have at least one `NaN` in any row by setting the `axis` argument to `1`:

```
>>> df.dropna(axis=1)
      A      B
0   1.0    2.0
1   5.0    6.0
2  10.0   11.0
```

The `dropna` method supports several additional parameters that can come in handy:

```
# only drop rows where all columns are NaN
# (returns the whole array here since we don't
# have a row with all values NaN)
>>> df.dropna(how='all')
      A     B     C     D
0   1.0   2.0   3.0   4.0
1   5.0   6.0   NaN   8.0
2  10.0  11.0  12.0   NaN


# drop rows that have fewer than 4 real values
>>> df.dropna(thresh=4)
      A     B     C     D
0   1.0   2.0   3.0   4.0


# only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
      A     B     C     D
0   1.0   2.0   3.0   4.0
2  10.0  11.0  12.0   NaN
```

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many samples, which will make a reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

Imputing missing values

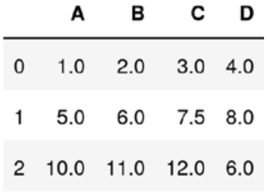
Often, the removal of training examples or dropping of entire feature columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training examples in our dataset. One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the `SimpleImputer` class from scikit-learn, as shown in the following code:

```
>>> from sklearn.impute import SimpleImputer
>>> import numpy as np
>>> imr = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imr = imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data

array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  7.5,   8.],
       [ 10.,  11.,  12.,   6.]])
```

Here, we replaced each `NaN` value with the corresponding mean, which is separately calculated for each feature column. Other options for the `strategy` parameter are `median` or `most_frequent`, where the latter replaces the missing values with the most frequent values. This is useful for imputing categorical feature values, for example, a feature column that stores an encoding of color names, such as red, green, and blue. We will encounter examples of such data later in this chapter.

Alternatively, an even more convenient way to impute missing values is by using pandas' `fillna` method and providing an imputation method as an argument. For example, using pandas, we could achieve the same mean imputation directly in the `DataFrame` object via the following command:

```
>>> df.fillna(df.mean())
```

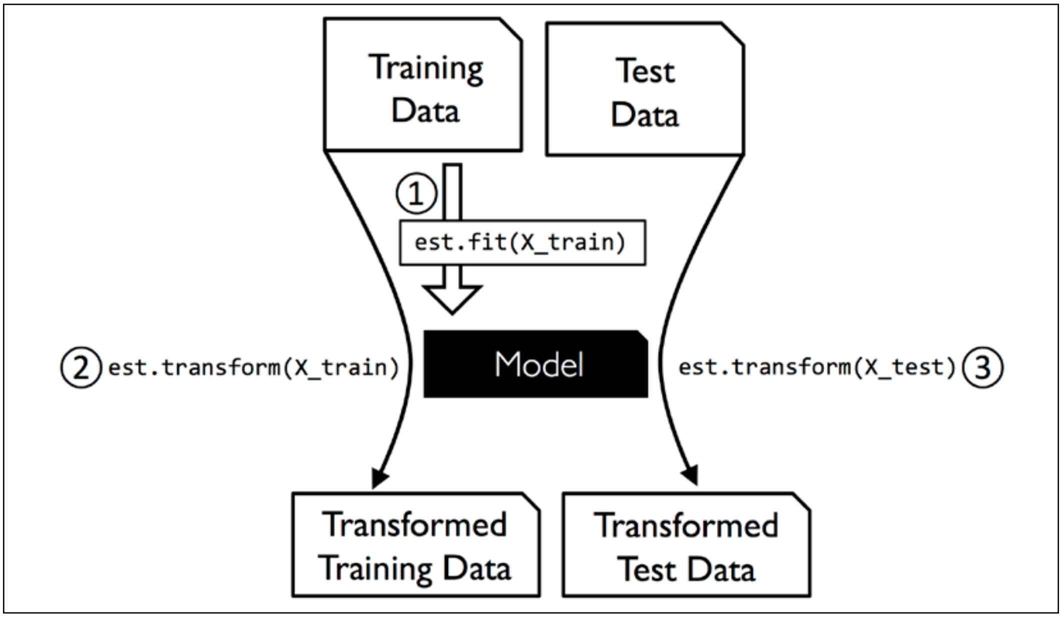


	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

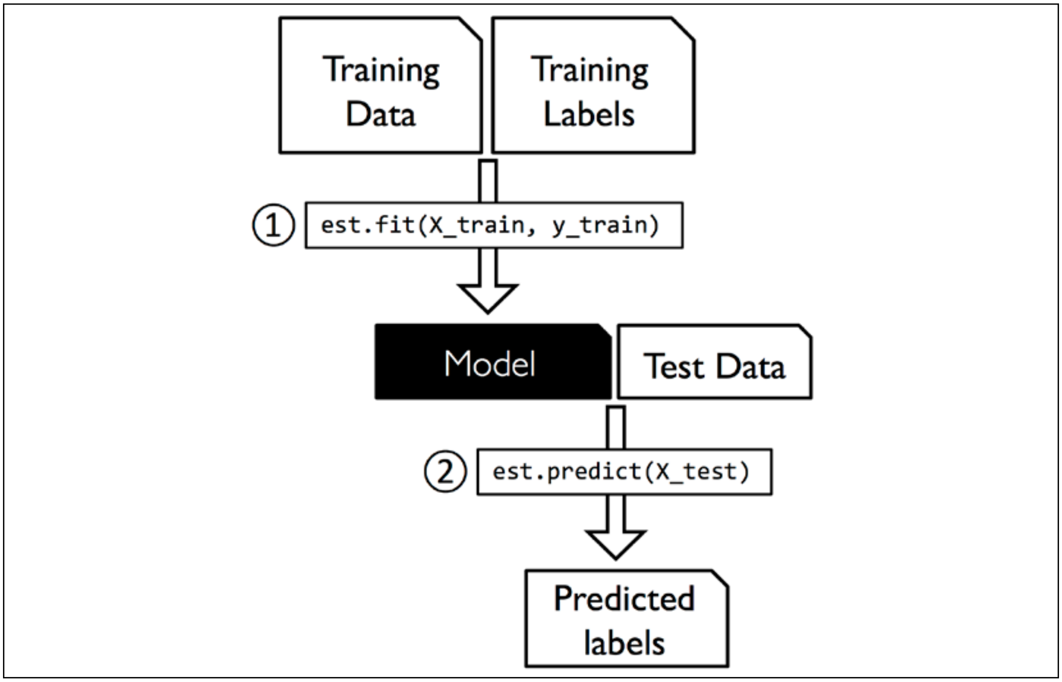
Understanding the scikit-learn estimator API

In the previous section, we used the `SimpleImputer` class from scikit-learn to impute missing values in our dataset. The `SimpleImputer` class belongs to the so-called **transformer** classes in scikit-learn, which are used for data transformation. The two essential methods of those estimators are `fit` and `transform`. The `fit` method is used to learn the parameters from the training data, and the `transform` method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model.

The following figure illustrates how a transformer, fitted on the training data, is used to transform a training dataset as well as a new test dataset:



The classifiers that we used in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, belong to the so-called **estimators** in scikit-learn, with an API that is conceptually very similar to the transformer class. Estimators have a `predict` method but can also have a `transform` method, as you will see later in this chapter. As you may recall, we also used the `fit` method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new, unlabeled data examples via the `predict` method, as illustrated in the following figure:



Handling categorical data

So far, we have only been working with numerical values. However, it is not uncommon for real-world datasets to contain one or more categorical feature columns. In this section, we will make use of simple yet effective examples to see how to deal with this type of data in numerical computing libraries.

When we are talking about categorical data, we have to further distinguish between **ordinal** and **nominal** features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order: *XL* > *L* > *M*. In contrast, nominal features don't imply any order and, to continue with the previous example, we could think of t-shirt color as a nominal feature since it typically doesn't make sense to say that, for example, red is larger than blue.

Categorical data encoding with pandas

Before we explore different techniques for handling such categorical data, let's create a new `DataFrame` to illustrate the problem:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...            ['green', 'M', 10.1, 'class2'],
...            ['red', 'L', 13.5, 'class1'],
...            ['blue', 'XL', 15.3, 'class2']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
    color  size  price  classlabel
0   green     M   10.1      class2
1     red     L   13.5      class1
2    blue    XL   15.3      class2
```

As we can see in the preceding output, the newly created `DataFrame` contains a nominal feature (`color`), an ordinal feature (`size`), and a numerical feature (`price`) column. The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column. The learning algorithms for classification that we discuss in this book do not use ordinal information in class labels.

Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our `size` feature, so we have to define the mapping manually. In the following simple example, let's assume that we know the numerical difference between features, for example, $XL = L + 1 = M + 2$:

```
>>> size_mapping = {'XL': 3,
...                 'L': 2,
...                 'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
    color  size  price  classlabel
0   green     1   10.1      class2
1     red     2   13.5      class1
2    blue     3   15.3      class2
```

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary, `inv_size_mapping = {v: k for k, v in size_mapping.items()}`, which can then be used via the pandas `map` method on the transformed feature column and is similar to the `size_mapping` dictionary that we used previously. We can use it as follows:

```
>>> inv_size_mapping = {v: k for k, v in size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0   M
1   L
2   XL
Name: size, dtype: object
```

Encoding class labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are *not* ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels, starting at `0`:

```
>>> import numpy as np
>>> class_mapping = {label: idx for idx, label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Next, we can use the mapping dictionary to transform the class labels into integers:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
    color  size  price  classlabel
0   green     1   10.1           1
1     red     2   13.5           0
2    blue     3   15.3           1
```

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
```

```
      color  size  price  classlabel
   0  green     1   10.1      class2
   1    red     2   13.5      class1
   2   blue     3   15.3      class2
```

Alternatively, there is a convenient `LabelEncoder` class directly implemented in scikit-learn to achieve this:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([1, 0, 1])
```

Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately, and we can use the `inverse_transform` method to transform the integer class labels back into their original string representation:

```
>>> class_le.inverse_transform(y)
array(['class2', 'class1', 'class2'], dtype=object)
```

Performing one-hot encoding on nominal features

In the previous *Mapping ordinal features* section, we used a simple dictionary-mapping approach to convert the ordinal `size` feature into integers. Since scikit-learn's estimators for classification treat class labels as categorical data that does not imply any order (nominal), we used the convenient `LabelEncoder` to encode the string labels into integers. It may appear that we could use a similar approach to transform the nominal `color` column of our dataset, as follows:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

After executing the preceding code, the first column of the NumPy array, `X`, now holds the new `color` values, which are encoded as follows:

- `blue = 0`
- `green = 1`
- `red = 2`

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, a learning algorithm will now assume that `green` is larger than `blue`, and `red` is larger than `green`. Although this assumption is incorrect, the algorithm could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called **one-hot encoding**. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column. Here, we would convert the `color` feature into three new features: `blue`, `green`, and `red`. Binary values can then be used to indicate the particular `color` of an example; for example, a `blue` example can be encoded as `blue=1`, `green=0`, `red=0`. To perform this transformation, we can use the `OneHotEncoder` that is implemented in scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> X = df[['color', 'size', 'price']].values
>>> color_ohe = OneHotEncoder()
>>> color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()

    array([[0., 1., 0.],
           [0., 0., 1.],
           [1., 0., 0.]])
```

Note that we applied the `OneHotEncoder` to only a single column, `(X[:, 0].reshape(-1, 1)))`, to avoid modifying the other two columns in the array as well. If we want to selectively transform columns in a multi-feature array, we can use the `ColumnTransformer`, which accepts a list of `(name, transformer, column(s))` tuples as follows:

```
>>> from sklearn.compose import ColumnTransformer
>>> X = df[['color', 'size', 'price']].values
>>> c_transf = ColumnTransformer([
...     ('onehot', OneHotEncoder(), [0]),
...     ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
    array([[0.0, 1.0, 0.0, 1, 10.1],
           [0.0, 0.0, 1.0, 2, 13.5],
           [1.0, 0.0, 0.0, 3, 15.3]])
```

In the preceding code example, we specified that we want to modify only the first column and leave the other two columns untouched via the `'passthrough'` argument.

An even more convenient way to create those dummy features via one-hot encoding is to use the get_dummies method implemented in pandas. Applied to a DataFrame, the get_dummies method will only convert string columns and leave all other columns unchanged:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
    price  size  color_blue  color_green  color_red
0    10.1     1           0            1          0
1    13.5     2           0            0          1
2    15.3     3           1            0          0
```

When we are using one-hot encoding datasets, we have to keep in mind that this introduces multicollinearity, which can be an issue for certain methods (for instance, methods that require matrix inversion). If features are highly correlated, matrices are computationally difficult to invert, which can lead to numerically unstable estimates. To reduce the correlation among variables, we can simply remove one feature column from the one-hot encoded array. Note that we do not lose any important information by removing a feature column, though; for example, if we remove the column color_blue, the feature information is still preserved since if we observe color_green=0 and color_red=0, it implies that the observation must be blue.

If we use the get_dummies function, we can drop the first column by passing a True argument to the drop_first parameter, as shown in the following code example:

```
>>> pd.get_dummies(df[['price', 'color', 'size']],
...                drop_first=True)
    price  size  color_green  color_red
0    10.1     1            1          0
1    13.5     2            0          1
2    15.3     3            0          0
```

In order to drop a redundant column via the OneHotEncoder, we need to set drop='first' and set categories='auto' as follows:

```
>>> color_ohe = OneHotEncoder(categories='auto', drop='first')
>>> c_transf = ColumnTransformer([
...            ('onehot', color_ohe, [0]),
...            ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)

array([[  1. ,  0. ,  1. ,  10.1],
       [  0. ,  1. ,  2. ,  13.5],
       [  0. ,  0. ,  3. ,  15.3]])
```

Optional: encoding ordinal features

If we are unsure about the numerical differences between the categories of ordinal features, or the difference between two ordinal values is not defined, we can also encode them using a threshold encoding with 0/1 values. For example, we can split the feature `size` with values M, L, and XL into two new features, "x > M" and "x > L". Let's consider the original DataFrame:

```
>>> df = pd.DataFrame([['green', 'M', 10.1,
...                     'class2'],
...                    ['red', 'L', 13.5,
...                     'class1'],
...                    ['blue', 'XL', 15.3,
...                     'class2']])

>>> df.columns = ['color', 'size', 'price',
...               'classlabel']
>>> df
```

We can use the apply method of pandas' DataFrames to write custom lambda expressions in order to encode these variables using the value-threshold approach:

```
>>> df['x > M'] = df['size'].apply(
...     lambda x: 1 if x in {'L', 'XL'} else 0)
>>> df['x > L'] = df['size'].apply(
...     lambda x: 1 if x == 'XL' else 0)
>>> del df['size']
>>> df
```

Partitioning a dataset into separate training and test datasets

We briefly introduced the concept of partitioning a dataset into separate datasets for training and testing in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, and *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*. Remember that comparing predictions to true labels in the test set can be understood as the unbiased performance evaluation of our model before we let it loose on the real world. In this section, we will prepare a new dataset, the **Wine** dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset.

The Wine dataset is another open source dataset that is available from the UCI machine learning repository (`https://archive.ics.uci.edu/ml/datasets/Wine`); it consists of 178 wine examples with 13 features describing their different chemical properties.

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the dataset at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable on the UCI server. For instance, to load the Wine dataset from a local directory, you can replace this line:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None)
```

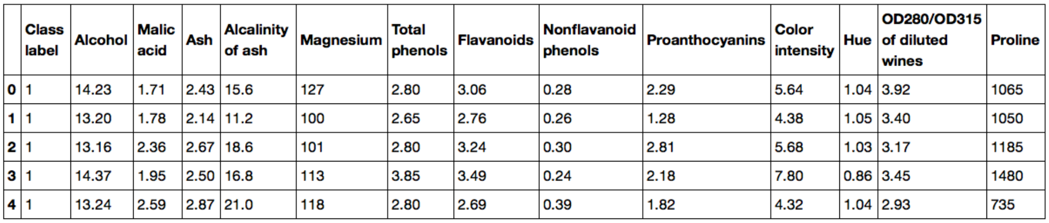
with the following one:

```
df = pd.read_csv(
    'your/local/path/to/wine.data', header=None)
```

Using the pandas library, we will directly read in the open source Wine dataset from the UCI machine learning repository:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/'
...                       'ml/machine-learning-databases/'
...                       'wine/wine.data', header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

The 13 different features in the Wine dataset, describing the chemical properties of the 178 wine examples, are listed in the following table:



	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

The examples belong to one of three different classes, `1`, `2`, and `3`, which refer to the three different types of grape grown in the same region in Italy but derived from different wine cultivars, as described in the dataset summary (`https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names`).

A convenient way to randomly partition this dataset into separate test and training datasets is to use the `train_test_split` function from scikit-learn's `model_selection` submodule:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.3,
...                      random_state=0,
...                      stratify=y)
```

First, we assigned the NumPy array representation of the feature columns 1-13 to the variable `X` and we assigned the class labels from the first column to the variable `y`. Then, we used the `train_test_split` function to randomly split `X` and `y` into separate training and test datasets. By setting `test_size=0.3`, we assigned 30 percent of the wine examples to `X_test` and `y_test`, and the remaining 70 percent of the examples were assigned to `X_train` and `y_train`, respectively. Providing the class label array `y` as an argument to `stratify` ensures that both training and test datasets have the same class proportions as the original dataset.

Choosing an appropriate ratio for partitioning a dataset into training and test datasets

If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information to the test set. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test datasets is all about balancing this tradeoff. In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits are also common and appropriate. For example, if the dataset contains more than 100,000 training examples, it might be fine to withhold only 10,000 examples for testing in order to get a good estimate of the generalization performance. More information and illustrations can be found in section one of my article *Model evaluation, model selection, and algorithm selection in machine learning*, which is freely available at `https://arxiv.org/pdf/1811.12808.pdf`.

Moreover, instead of discarding the allocated test data after model training and evaluation, it is a common practice to retrain a classifier on the entire dataset, as it can improve the predictive performance of the model. While this approach is generally recommended, it could lead to worse generalization performance if the dataset is small and the test dataset contains outliers, for example. Also, after refitting the model on the whole dataset, we don't have any independent data left to evaluate its performance.

Bringing features onto the same scale

Feature scaling is a crucial step in our preprocessing pipeline that can easily be forgotten. **Decision trees** and **random forests** are two of the very few machine learning algorithms where we don't need to worry about feature scaling. Those algorithms are scale invariant. However, the majority of machine learning and optimization algorithms behave much better if features are on the same scale, as we saw in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, when we implemented the **gradient descent optimization** algorithm.

The importance of feature scaling can be illustrated by a simple example. Let's assume that we have two features where one feature is measured on a scale from 1 to 10 and the second feature is measured on a scale from 1 to 100,000, respectively.

When we think of the squared error function in Adaline from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, it makes sense to say that the algorithm will mostly be busy optimizing the weights according to the larger errors in the second feature. Another example is the **k-nearest neighbors** (**KNN**) algorithm with a Euclidean distance measure: the computed distances between examples will be dominated by the second feature axis.

Now, there are two common approaches to bringing different features onto the same scale: **normalization** and **standardization**. Those terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, normalization refers to the rescaling of the features to a range of [0, 1], which is a special case of **min-max scaling**. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value, $x_{norm}^{(i)}$, of an example, $x^{(i)}$, can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Here, $x^{(i)}$ is a particular example, x_{min} is the smallest value in a feature column, and x_{max} is the largest value.

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, standardization can be more practical for many machine learning algorithms, especially for optimization algorithms such as gradient descent. The reason is that many linear models, such as the logistic regression and SVM from *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, initialize the weights to 0 or small random values close to 0. Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns have the same parameters as a standard normal distribution (zero mean and unit variance), which makes it easier to learn the weights. Furthermore, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

The procedure for standardization can be expressed by the following equation:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column, and σ_x is the corresponding standard deviation.

The following table illustrates the difference between the two commonly used feature scaling techniques, standardization and normalization, on a simple example dataset consisting of numbers 0 to 5:

Input	**Standardized**	**Min-max normalized**
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

You can perform the standardization and normalization shown in the table manually by executing the following code examples:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('standardized:', (ex - ex.mean()) / ex.std())
standardized: [-1.46385011  -0.87831007  -0.29277002  0.29277002
0.87831007  1.46385011]
>>> print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
normalized: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Similar to the `MinMaxScaler` class, scikit-learn also implements a class for standardization:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Again, it is also important to highlight that we fit the `StandardScaler` class only once—on the training data—and use those parameters to transform the test dataset or any new data point.

Other, more advanced methods for feature scaling are available from scikit-learn, such as the `RobustScaler`. The `RobustScaler` is especially helpful and recommended if we are working with small datasets that contain many outliers. Similarly, if the machine learning algorithm applied to this dataset is prone to **overfitting**, the `RobustScaler` can be a good choice. Operating on each feature column independently, the `RobustScaler` removes the median value and scales the dataset according to the 1st and 3rd quartile of the dataset (that is, the 25th and 75th quantile, respectively) such that more extreme values and outliers become less pronounced. The interested reader can find more information about the `RobustScaler` in the official scikit-learn documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html`.

Selecting meaningful features

If we notice that a model performs much better on a training dataset than on the test dataset, this observation is a strong indicator of overfitting. As we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, overfitting means the model fits the parameters too closely with regard to the particular observations in the training dataset, but does not generalize well to new data; we say that the model has a **high variance**. The reason for the overfitting is that our model is too complex for the given training data. Common solutions to reduce the generalization error are as follows:

- Collect more training data
- Introduce a penalty for complexity via regularization
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data

Collecting more training data is often not applicable. In *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will learn about a useful technique to check whether more training data is helpful. In the following sections, we will look at common ways to reduce overfitting by regularization and dimensionality reduction via feature selection, which leads to simpler models by requiring fewer parameters to be fitted to the data.

L1 and L2 regularization as penalties against model complexity

You will recall from *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, that **L2 regularization** is one approach to reduce the complexity of a model by penalizing large individual weights. We defined the squared L2 norm of our weight vector, $\boldsymbol{w}$, as follows:

$$L2: \quad \|\boldsymbol{w}\|_2^2 \;=\; \sum_{j=1}^{m} w_j^2$$

Another approach to reduce the model complexity is the related **L1 regularization**:

$$L1: \quad \|\boldsymbol{w}\|_1 \;=\; \sum_{j=1}^{m} \left|w_j\right|$$

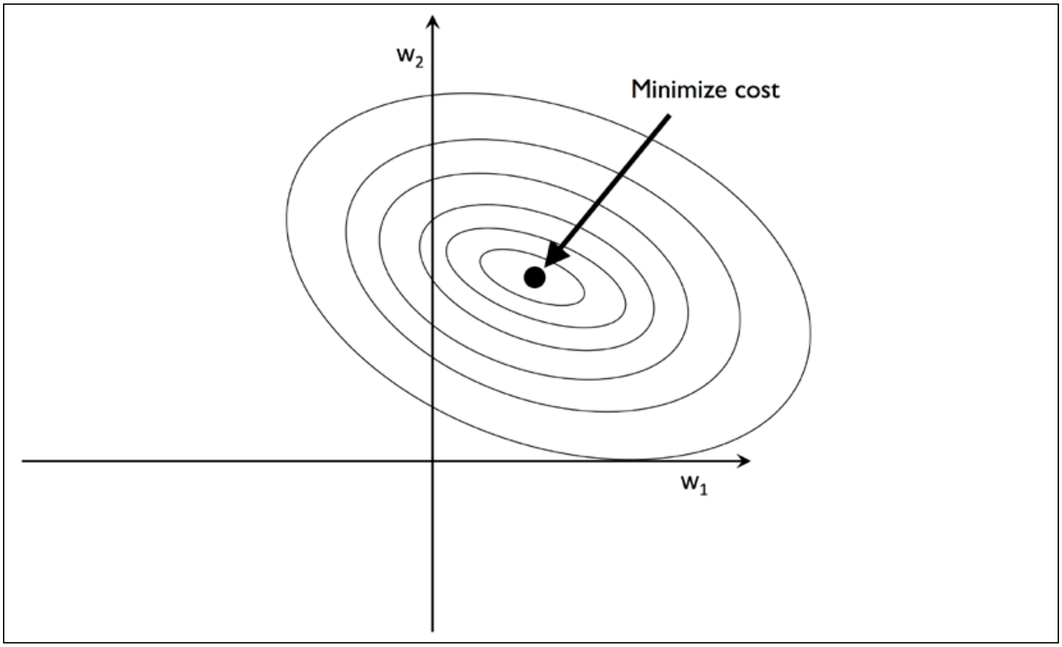
Here, we simply replaced the square of the weights by the sum of the absolute values of the weights. In contrast to L2 regularization, L1 regularization usually yields sparse feature vectors and most feature weights will be zero. Sparsity can be useful in practice if we have a high-dimensional dataset with many features that are irrelevant, especially in cases where we have more irrelevant dimensions than training examples. In this sense, L1 regularization can be understood as a technique for feature selection.

A geometric interpretation of L2 regularization

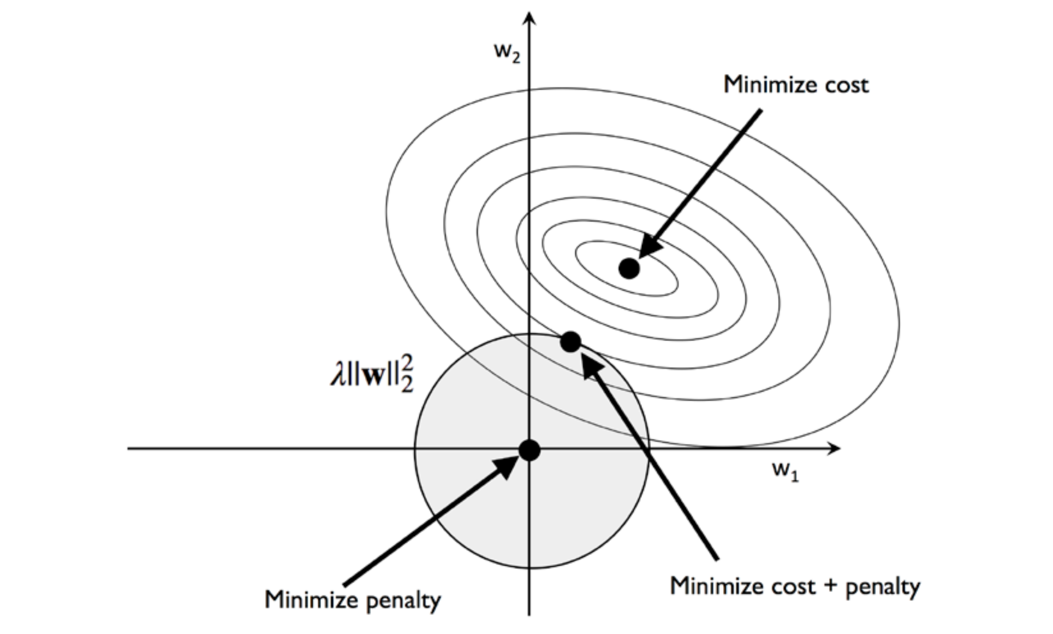
As mentioned in the previous section, L2 regularization adds a penalty term to the cost function that effectively results in less extreme weight values compared to a model trained with an unregularized cost function.

To better understand how L1 regularization encourages sparsity, let's take a step back and take a look at a geometric interpretation of regularization. Let's plot the contours of a convex cost function for two weight coefficients, w_1 and w_2.

Here, we will consider the **sum of squared errors** (**SSE**) cost function that we used for Adaline in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, since it is spherical and easier to draw than the cost function of logistic regression; however, the same concepts apply. Remember that our goal is to find the combination of weight coefficients that minimize the cost function for the training data, as shown in the following figure (the point in the center of the ellipses):



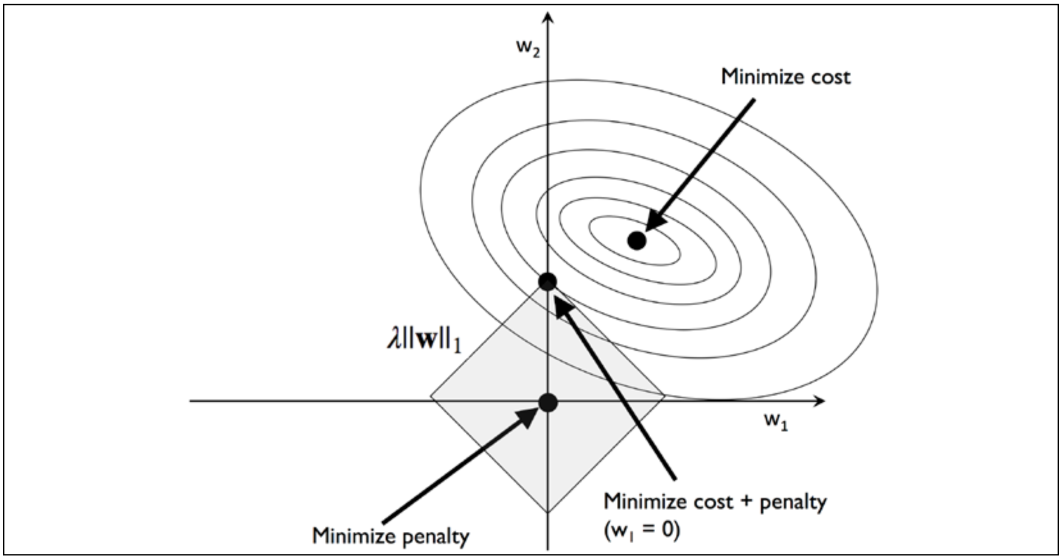
We can think of regularization as adding a penalty term to the cost function to encourage smaller weights; in other words, we penalize large weights. Thus, by increasing the regularization strength via the regularization parameter, λ, we shrink the weights toward zero and decrease the dependence of our model on the training data. Let's illustrate this concept in the following figure for the L2 penalty term:



The quadratic L2 regularization term is represented by the shaded ball. Here, our weight coefficients cannot exceed our regularization budget—the combination of the weight coefficients cannot fall outside the shaded area. On the other hand, we still want to minimize the cost function. Under the penalty constraint, our best effort is to choose the point where the L2 ball intersects with the contours of the unpenalized cost function. The larger the value of the regularization parameter, λ, gets, the faster the penalized cost grows, which leads to a narrower L2 ball. For example, if we increase the regularization parameter towards infinity, the weight coefficients will become effectively zero, denoted by the center of the L2 ball. To summarize the main message of the example, our goal is to minimize the sum of the unpenalized cost plus the penalty term, which can be understood as adding bias and preferring a simpler model to reduce the variance in the absence of sufficient training data to fit the model.

Sparse solutions with L1 regularization

Now, let's discuss L1 regularization and sparsity. The main concept behind L1 regularization is similar to what we discussed in the previous section. However, since the L1 penalty is the sum of the absolute weight coefficients (remember that the L2 term is quadratic), we can represent it as a diamond-shape budget, as shown in the following figure:



In the preceding figure, we can see that the contour of the cost function touches the L1 diamond at $w_1 = 0$. Since the contours of an L1 regularized system are sharp, it is more likely that the optimum—that is, the intersection between the ellipses of the cost function and the boundary of the L1 diamond—is located on the axes, which encourages sparsity.

L1 regularization and sparsity

The mathematical details of why L1 regularization can lead to sparse solutions are beyond the scope of this book. If you are interested, an excellent explanation of L2 versus L1 regularization can be found in *Section 3.4*, *The Elements of Statistical Learning*, *Trevor Hastie*, *Robert Tibshirani*, and *Jerome Friedman*, *Springer Science+Business Media, 2009*.

For regularized models in scikit-learn that support L1 regularization, we can simply set the `penalty` parameter to `'l1'` to obtain a sparse solution:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1',
...                    solver='liblinear',
...                    multi_class='ovr')
```

Note that we also need to select a different optimization algorithm (for example, `solver='liblinear'`), since `'lbfgs'` currently does not support L1-regularized loss optimization. Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```
>>> lr = LogisticRegression(penalty='l1',
...                         C=1.0,
...                         solver='liblinear',
...                         multi_class='ovr')
# Note that C=1.0 is the default. You can increase
# or decrease it to make the regularization effect
# stronger or weaker, respectively.
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 1.0
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 1.0
```

Both training and test accuracies (both 100 percent) indicate that our model does a perfect job on both datasets. When we access the intercept terms via the `lr.intercept_` attribute, we can see that the array returns three values:

```
>>> lr.intercept_
array([-1.26346036, -1.21584018, -2.3697841 ])
```

Since we fit the `LogisticRegression` object on a multiclass dataset via the **one-vs.-rest** (**OvR**) approach, the first intercept belongs to the model that fits class 1 versus classes 2 and 3, the second value is the intercept of the model that fits class 2 versus classes 1 and 3, and the third value is the intercept of the model that fits class 3 versus classes 1 and 2:

```
>>> lr.coef_
array([[ 1.24590762,  0.18070219,  0.74375939, -1.16141503,
         0.        ,  0.        ,  1.16926815,  0.        ,
         0.        ,  0.        ,  0.        ,  0.54784923,
         2.51028042],
       [-1.53680415, -0.38795309, -0.99494046,  0.36508729,
        -0.05981561,  0.        ,  0.6681573 ,  0.        ,
         0.        , -1.93426485,  1.23265994,  0.        ,
          -2.23137595],
         [ 0.13547047,  0.16873019,  0.35728003,  0.        ,
           0.        ,  0.        , -2.43713947,  0.        ,
           0.        ,  1.56351492, -0.81894749, -0.49308407,
           0.        ]])
```

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights, where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:

$$z = w_0x_0 + \cdots + w_mx_m = \sum_{j=0}^{m} x_jw_j = \boldsymbol{w}^T\boldsymbol{x}$$

Accessing the bias and weight parameters of scikit-learn estimators

In scikit-learn, the `intercept_` corresponds to w_0 and `coef_` corresponds to the values w_j for $j > 0$.

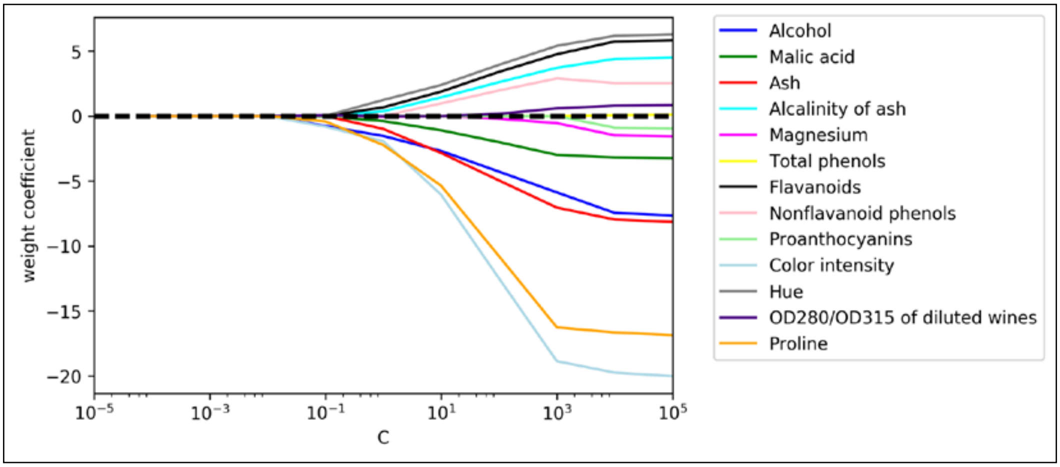
As a result of L1 regularization, which, as mentioned, serves as a method for feature selection, we just trained a model that is robust to the potentially irrelevant features in this dataset. Strictly speaking, though, the weight vectors from the previous example are not necessarily sparse because they contain more non-zero than zero entries. However, we could enforce sparsity (more zero entries) by further increasing the regularization strength—that is, choosing lower values for the `C` parameter.

In the last example on regularization in this chapter, we will vary the regularization strength and plot the regularization path—the weight coefficients of the different features for different regularization strengths:

```
>>> import matplotlib.pyplot as plt

>>> fig = plt.figure()
>>> ax = plt.subplot(111)

>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4., 6.):
```

```
...     lr = LogisticRegression(penalty='l1', C=10.**c,
...                             solver='liblinear',
...                             multi_class='ovr', random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)

>>> weights = np.array(weights)

>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column + 1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('weight coefficient')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

The resulting plot provides us with further insights into the behavior of L1 regularization. As we can see, all feature weights will be zero if we penalize the model with a strong regularization parameter ($C < 0.01$); C is the inverse of the regularization parameter, λ:



Sequential feature selection algorithms

An alternative way to reduce the complexity of the model and avoid overfitting is **dimensionality reduction** via feature selection, which is especially useful for unregularized models. There are two main categories of dimensionality reduction techniques: **feature selection** and **feature extraction**. Via feature selection, we select a subset of the original features, whereas in feature extraction, we derive information from the feature set to construct a new feature subspace.

In this section, we will take a look at a classic family of feature selection algorithms. In the next chapter, *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will learn about different feature extraction techniques to compress a dataset onto a lower-dimensional feature subspace.

Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial *d*-dimensional feature space to a *k*-dimensional feature subspace where *k*<*d*. The motivation behind feature selection algorithms is to automatically select a subset of features that are most relevant to the problem, to improve computational efficiency, or to reduce the generalization error of the model by removing irrelevant features or noise, which can be useful for algorithms that don't support regularization.

A classic sequential feature selection algorithm is **sequential backward selection** (**SBS**), which aims to reduce the dimensionality of the initial feature subspace with a minimum decay in the performance of the classifier to improve upon computational efficiency. In certain cases, SBS can even improve the predictive power of the model if a model suffers from overfitting.

> **Greedy search algorithms**
>
> **Greedy algorithms** make locally optimal choices at each stage of a combinatorial search problem and generally yield a suboptimal solution to the problem, in contrast to **exhaustive search algorithms**, which evaluate all possible combinations and are guaranteed to find the optimal solution. However, in practice, an exhaustive search is often computationally not feasible, whereas greedy algorithms allow for a less complex, computationally more efficient solution.

The idea behind the SBS algorithm is quite simple: SBS sequentially removes features from the full feature subset until the new feature subspace contains the desired number of features. In order to determine which feature is to be removed at each stage, we need to define the criterion function, *J*, that we want to minimize.

The criterion calculated by the criterion function can simply be the difference in performance of the classifier before and after the removal of a particular feature. Then, the feature to be removed at each stage can simply be defined as the feature that maximizes this criterion; or in more simple terms, at each stage we eliminate the feature that causes the least performance loss after removal. Based on the preceding definition of SBS, we can outline the algorithm in four simple steps:

1. Initialize the algorithm with *k* = *d*, where *d* is the dimensionality of the full feature space, $\boldsymbol{X}_d$.
2. Determine the feature, $\boldsymbol{x}^-$, that maximizes the criterion: $\boldsymbol{x}^- = \operatorname{argmax} J(\boldsymbol{X}_k - \boldsymbol{x})$, where $\boldsymbol{x} \in \boldsymbol{X}_k$.
3. Remove the feature, $\boldsymbol{x}^-$, from the feature set: $\boldsymbol{X}_{k-1} = \boldsymbol{X}_k - \boldsymbol{x}^-\,;\ k = k - 1$.
4. Terminate if *k* equals the number of desired features; otherwise, go to step 2.

A resource on sequential feature algorithms

You can find a detailed evaluation of several sequential feature algorithms in *Comparative Study of Techniques for Large-Scale Feature Selection*, *F. Ferri*, *P. Pudil*, *M. Hatef*, and *J. Kittler*, pages 403-413, *1994*.

Unfortunately, the SBS algorithm has not been implemented in scikit-learn yet. But since it is so simple, let's go ahead and implement it in Python from scratch:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split


class SBS():
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

In the preceding implementation, we defined the `k_features` parameter to specify the desired number of features we want to return. By default, we use the `accuracy_score` from scikit-learn to evaluate the performance of a model (an estimator for classification) on the feature subsets.

Inside the while loop of the fit method, the feature subsets created by the itertools.combination function are evaluated and reduced until the feature subset has the desired dimensionality. In each iteration, the accuracy score of the best subset is collected in a list, self.scores_, based on the internally created test dataset, X_test. We will use those scores later to evaluate the results. The column indices of the final feature subset are assigned to self.indices_, which we can use via the transform method to return a new data array with the selected feature columns. Note that, instead of calculating the criterion explicitly inside the fit method, we simply removed the feature that is not contained in the best performing feature subset.

Now, let's see our SBS implementation in action using the KNN classifier from scikit-learn:

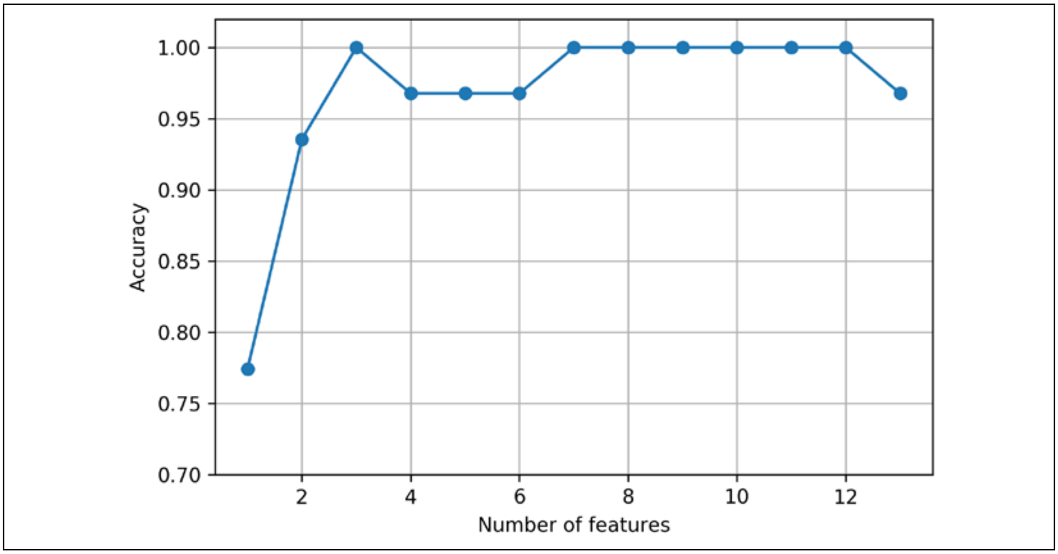
```
>>> import matplotlib.pyplot as plt
>>> from sklearn.neighbors import KNeighborsClassifier

>>> knn = KNeighborsClassifier(n_neighbors=5)

>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Although our SBS implementation already splits the dataset into a test and training dataset inside the fit function, we still fed the training dataset, X_train, to the algorithm. The SBS fit method will then create new training subsets for testing (validation) and training, which is why this test set is also called the **validation dataset**. This approach is necessary to prevent our *original* test set from becoming part of the training data.

Remember that our SBS algorithm collects the scores of the best feature subset at each stage, so let's move on to the more exciting part of our implementation and plot the classification accuracy of the KNN classifier that was calculated on the validation dataset. The code is as follows:

```
>>> k_feat = [len(k) for k in sbs.subsets_]

>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.02])
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the following figure, the accuracy of the KNN classifier improved on the validation dataset as we reduced the number of features, which is likely due to a decrease in the **curse of dimensionality** that we discussed in the context of the KNN algorithm in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*. Also, we can see in the following plot that the classifier achieved 100 percent accuracy for *k*={*3, 7, 8, 9, 10, 11, 12*}:



To satisfy our own curiosity, let's see what the smallest feature subset (*k*=*3*), which yielded such a good performance on the validation dataset, looks like:

```
>>> k3 = list(sbs.subsets_[10])
>>> print(df_wine.columns[1:][k3])
Index(['Alcohol', 'Malic acid', 'OD280/OD315 of diluted wines'],
dtype='object')
```

Using the preceding code, we obtained the column indices of the three-feature subset from the 11th position in the `sbs.subsets_` attribute and returned the corresponding feature names from the column index of the pandas Wine `DataFrame`.

Next, let's evaluate the performance of the KNN classifier on the original test dataset:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.967741935484
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.962962962963
```

In the preceding code section, we used the complete feature set and obtained approximately 97 percent accuracy on the training dataset and approximately 96 percent accuracy on the test dataset, which indicates that our model already generalizes well to new data. Now, let's use the selected three-feature subset and see how well KNN performs:

```
>>> knn.fit(X_train_std[:, k3], y_train)
>>> print('Training accuracy:',
...       knn.score(X_train_std[:, k3], y_train))
Training accuracy: 0.951612903226
>>> print('Test accuracy:',
...       knn.score(X_test_std[:, k3], y_test))
Test accuracy: 0.925925925926
```

When using less than a quarter of the original features in the Wine dataset, the prediction accuracy on the test dataset declined slightly. This may indicate that those three features do not provide less discriminatory information than the original dataset. However, we also have to keep in mind that the Wine dataset is a small dataset and is very susceptible to randomness—that is, the way we split the dataset into training and test subsets, and how we split the training dataset further into a training and validation subset.

While we did not increase the performance of the KNN model by reducing the number of features, we shrank the size of the dataset, which can be useful in real-world applications that may involve expensive data collection steps. Also, by substantially reducing the number of features, we obtain simpler models, which are easier to interpret.

Feature selection algorithms in scikit-learn

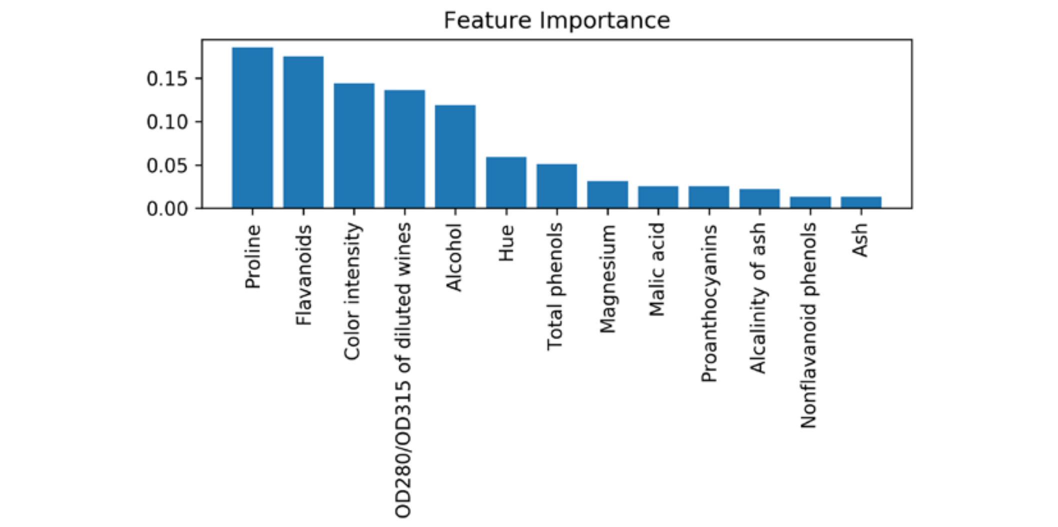
There are many more feature selection algorithms available via scikit-learn. Those include **recursive backward elimination** based on feature weights, tree-based methods to select features by importance, and univariate statistical tests. A comprehensive discussion of the different feature selection methods is beyond the scope of this book, but a good summary with illustrative examples can be found at `http://scikit-learn.org/stable/modules/feature_selection.html`. You can find implementations of several different flavors of sequential feature selection related to the simple SBS that we implemented previously in the Python package `mlxtend` at `http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/`.

Assessing feature importance with random forests

In previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and how to use the SBS algorithm for feature selection and apply it to a KNN algorithm. Another useful approach for selecting relevant features from a dataset is using a **random forest**, an ensemble technique that was introduced in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*. Using a random forest, we can measure the feature importance as the averaged impurity decrease computed from all decision trees in the forest, without making any assumptions about whether our data is linearly separable or not. Conveniently, the random forest implementation in scikit-learn already collects the feature importance values for us so that we can access them via the `feature_importances_` attribute after fitting a `RandomForestClassifier`. By executing the following code, we will now train a forest of 500 trees on the Wine dataset and rank the 13 features by their respective importance measures—remember from our discussion in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn* that we don't need to use standardized or normalized features in tree-based models:

```
>>> from sklearn.ensemble import RandomForestClassifier

>>> feat_labels = df_wine.columns[1:]

>>> forest = RandomForestClassifier(n_estimators=500,
...                                 random_state=1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_

>>> indices = np.argsort(importances)[::-1]

>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
>>> plt.title('Feature Importance')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         align='center')

>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels[indices] rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])

>>> plt.tight_layout()
>>> plt.show()
 1) Proline                         0.185453
 2) Flavanoids                      0.174751
 3) Color intensity                 0.143920
 4) OD280/OD315 of diluted wines    0.136162
 5) Alcohol                         0.118529
 6) Hue                             0.058739
 7) Total phenols                   0.050872
 8) Magnesium                       0.031357
 9) Malic acid                      0.025648
 10) Proanthocyanins                0.025570
 11) Alcalinity of ash              0.022366
 12) Nonflavanoid phenols           0.013354
 13) Ash                            0.013279
```

After executing the code, we created a plot that ranks the different features in the Wine dataset by their relative importance; note that the feature importance values are normalized so that they sum up to 1.0:



We can conclude that the proline and flavonoid levels, the color intensity, the OD280/OD315 diffraction, and the alcohol concentration of wine are the most discriminative features in the dataset based on the average impurity decrease in the 500 decision trees. Interestingly, two of the top-ranked features in the plot are also in the three-feature subset selection from the SBS algorithm that we implemented in the previous section (alcohol concentration and OD280/OD315 of diluted wines).

However, as far as interpretability is concerned, the random forest technique comes with an important *gotcha* that is worth mentioning. If two or more features are highly correlated, one feature may be ranked very highly while the information on the other feature(s) may not be fully captured. On the other hand, we don't need to be concerned about this problem if we are merely interested in the predictive performance of a model rather than the interpretation of feature importance values.

To conclude this section about feature importance values and random forests, it is worth mentioning that scikit-learn also implements a `SelectFromModel` object that selects features based on a user-specified threshold after model fitting, which is useful if we want to use the `RandomForestClassifier` as a feature selector and intermediate step in a scikit-learn `Pipeline` object, which allows us to connect different preprocessing steps with an estimator, as you will see in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*. For example, we could set the `threshold` to `0.1` to reduce the dataset to the five most important features using the following code:

```
>>> from sklearn.feature_selection import SelectFromModel

>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>>> X_selected = sfm.transform(X_train)
>>> print('Number of features that meet this threshold',
...       'criterion:', X_selected.shape[1])

Number of features that meet this threshold criterion: 5

>>> for f in range(X_selected.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
 1) Proline                        0.185453
 2) Flavanoids                     0.174751
 3) Color intensity                0.143920
 4) OD280/OD315 of diluted wines   0.136162
 5) Alcohol                        0.118529
```

Summary

We started this chapter by looking at useful techniques to make sure that we handle missing data correctly. Before we feed data to a machine learning algorithm, we also have to make sure that we encode categorical variables correctly, and in this chapter, we saw how we can map ordinal and nominal feature values to integer representations.

Moreover, we briefly discussed L1 regularization, which can help us to avoid overfitting by reducing the complexity of a model. As an alternative approach to removing irrelevant features, we used a sequential feature selection algorithm to select meaningful features from a dataset.

In the next chapter, you will learn about yet another useful approach to dimensionality reduction: feature extraction. It allows us to compress features onto a lower-dimensional subspace, rather than removing features entirely as in feature selection.

5

Compressing Data via Dimensionality Reduction

In *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, you learned about the different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is **feature extraction**. In this chapter, you will learn about three fundamental techniques that will help you to summarize the information content of a dataset by transforming it onto a new feature subspace of lower dimensionality than the original one. Data compression is an important topic in machine learning, and it helps us to store and analyze the increasing amounts of data that are produced and collected in the modern age of technology.

In this chapter, we will cover the following topics:

- **Principal component analysis** (**PCA**) for unsupervised data compression
- **Linear discriminant analysis** (**LDA**) as a supervised dimensionality reduction technique for maximizing class separability
- Nonlinear dimensionality reduction via **kernel principal component analysis** (**KPCA**)

Unsupervised dimensionality reduction via principal component analysis

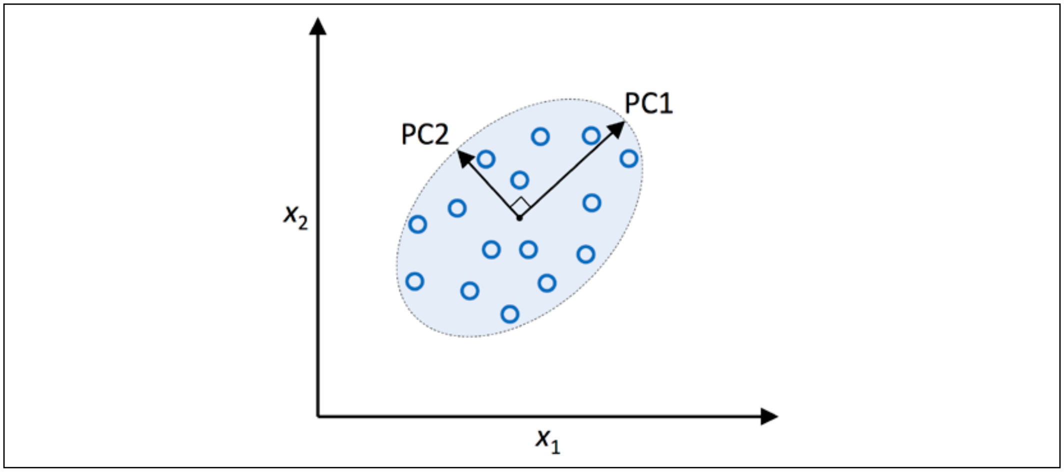
Similar to feature selection, we can use different feature extraction techniques to reduce the number of features in a dataset. The difference between feature selection and feature extraction is that while we maintain the original features when we use feature selection algorithms, such as **sequential backward selection**, we use feature extraction to transform or project the data onto a new feature space.

In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. In practice, feature extraction is not only used to improve storage space or the computational efficiency of the learning algorithm, but can also improve the predictive performance by reducing the **curse of dimensionality**—especially if we are working with non-regularized models.

The main steps behind principal component analysis

In this section, we will discuss PCA, an unsupervised linear transformation technique that is widely used across different fields, most prominently for feature extraction and dimensionality reduction. Other popular applications of PCA include exploratory data analyses and the denoising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics.

PCA helps us to identify patterns in data based on the correlation between features. In a nutshell, PCA aims to find the directions of maximum variance in high-dimensional data and projects the data onto a new subspace with equal or fewer dimensions than the original one. The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other, as illustrated in the following figure:



In the preceding figure, x_1 and x_2 are the original feature axes, and **PC1** and **PC2** are the principal components.

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix, $\boldsymbol{W}$, that allows us to map a vector, $\boldsymbol{x}$, the features of a training example, onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space. For instance, the process is as follows. Suppose we have a feature vector, $\boldsymbol{x}$:

$$\boldsymbol{x} = [x_1, x_2, \dots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d$$

which is then transformed by a transformation matrix, $\boldsymbol{W} \in \mathbb{R}^{d \times k}$:

$$\boldsymbol{xW} = \boldsymbol{z}$$

resulting in the output vector:

$$\boldsymbol{z} = [z_1, z_2, \dots, z_k], \quad \boldsymbol{z} \in \mathbb{R}^k$$

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically k << d), the first principal component will have the largest possible variance. All consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other principal components—even if the input features are correlated, the resulting principal components will be mutually orthogonal (uncorrelated). Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features *prior* to PCA if the features were measured on different scales and we want to assign equal importance to all features.

Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.
3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a projection matrix, $\boldsymbol{W}$, from the "top" k eigenvectors.

7. Transform the *d*-dimensional input dataset, ***X***, using the projection matrix, ***W***, to obtain the new *k*-dimensional feature subspace.

In the following sections, we will perform a PCA step by step, using Python as a learning exercise. Then, we will see how to perform a PCA more conveniently using scikit-learn.

Extracting the principal components step by step

In this subsection, we will tackle the first four steps of a PCA:

1. Standardizing the data.
2. Constructing the covariance matrix.
3. Obtaining the eigenvalues and eigenvectors of the covariance matrix.
4. Sorting the eigenvalues by decreasing order to rank the eigenvectors.

First, we will start by loading the Wine dataset that we were working with in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...           'machine-learning-databases/wine/wine.data',
...           header=None)
```

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable. For instance, to load the Wine dataset from a local directory, you can replace the following line:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None)
```

with the following one:

```
df = pd.read_csv(
    'your/local/path/to/wine.data',
    header=None)
```

Next, we will process the Wine data into separate training and test datasets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                      stratify=y,
...                      random_state=0)
>>> # standardize the features
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

After completing the mandatory preprocessing by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$ -dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features, x_j and x_k, on the population level can be calculated via the following equation:

$$\sigma_{jk} = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Here, μ_j and μ_k are the sample means of features *j* and *k*, respectively. Note that the sample means are zero if we standardized the dataset. A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions. For example, the covariance matrix of three features can then be written as follows (note that Σ stands for the Greek uppercase letter sigma, which is not to be confused with the summation symbol):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. In the case of the Wine dataset, we would obtain 13 eigenvectors and eigenvalues from the 13×13-dimensional covariance matrix.

Now, for our third step, let's obtain the eigenpairs of the covariance matrix. As you will remember from our introductory linear algebra classes, an eigenvector, $\boldsymbol{v}$, satisfies the following condition:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task, we will use the `linalg.eig` function from NumPy to obtain the eigenpairs of the Wine covariance matrix:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n%s' % eigen_vals)
Eigenvalues
[ 4.84274532  2.41602459  1.54845825  0.96120438  0.84166161
  0.6620634   0.51828472  0.34650377  0.3131368   0.10754642
  0.21357215  0.15362835  0.1808613 ]
```

Using the `numpy.cov` function, we computed the covariance matrix of the standardized training dataset. Using the `linalg.eig` function, we performed the eigendecomposition, which yielded a vector (`eigen_vals`) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13 × 13-dimensional matrix (`eigen_vecs`).

Eigendecomposition in NumPy

The `numpy.linalg.eig` function was designed to operate on both symmetric and non-symmetric square matrices. However, you may find that it returns complex eigenvalues in certain cases.

A related function, `numpy.linalg.eigh`, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to working with symmetric matrices such as the covariance matrix; `numpy.linalg.eigh` always returns real eigenvalues.

Total and explained variance

Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains most of the information (variance). The eigenvalues define the magnitude of the eigenvectors, so we have to sort the eigenvalues by decreasing magnitude; we are interested in the top *k* eigenvectors based on the values of their corresponding eigenvalues. But before we collect those *k* most informative eigenvectors, let's plot the **variance explained ratios** of the eigenvalues. The variance explained ratio of an eigenvalue, λ_j, is simply the fraction of an eigenvalue, λ_j, and the total sum of the eigenvalues:

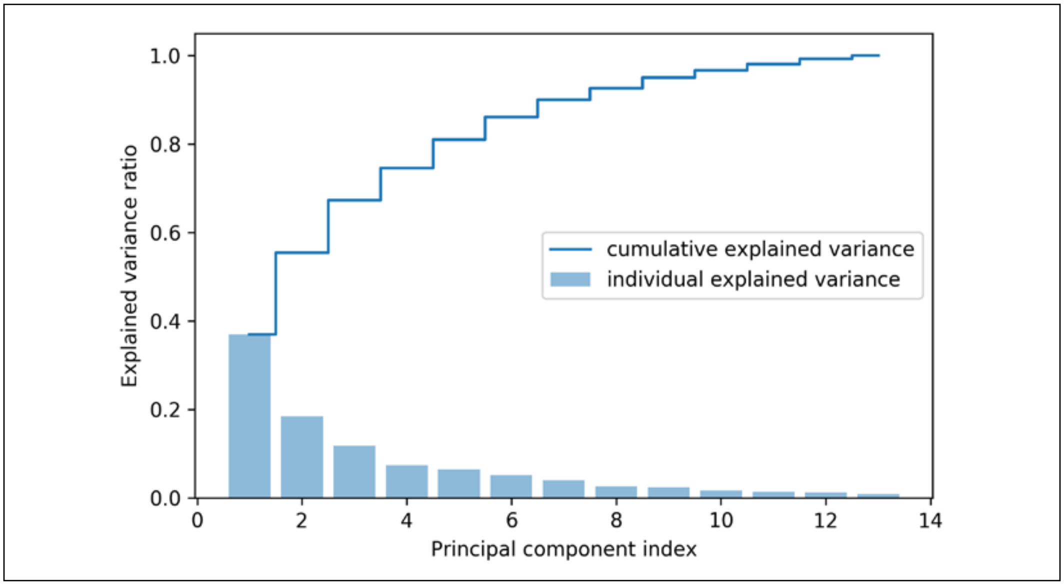
$$\text{Explained variance ratio} = \frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will then plot via Matplotlib's `step` function:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='Individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...          label='Cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal component index')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting plot indicates that the first principal component alone accounts for approximately 40 percent of the variance.

Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the dataset:



Although the explained variance plot reminds us of the feature importance values that we computed in *Chapter 4, Building Good Training Datasets – Data Preprocessing*, via random forests, we should remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored. Whereas a random forest uses the class membership information to compute the node impurities, variance measures the spread of values along a feature axis.

Feature transformation

Now that we have successfully decomposed the covariance matrix into eigenpairs, let's proceed with the last three steps to transform the Wine dataset onto the new principal component axes. The remaining steps we are going to tackle in this section are the following:

5. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a projection matrix, $\boldsymbol{W}$, from the "top" k eigenvectors.

7. Transform the *d*-dimensional input dataset, ***X***, using the projection matrix, ***W***, to obtain the new *k*-dimensional feature subspace.

Or, in less technical terms, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

We start by sorting the eigenpairs by decreasing order of the eigenvalues:

```
>>> # Make a list of (eigenvalue, eigenvector) tuples
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...                 for i in range(len(eigen_vals))]
>>> # Sort the (eigenvalue, eigenvector) tuples from high to low
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Next, we collect the two eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset. Note that two eigenvectors have been chosen for the purpose of illustration, since we are going to plot the data via a two-dimensional scatter plot later in this subsection. In practice, the number of principal components has to be determined by a tradeoff between computational efficiency and the performance of the classifier:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                 eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n', w)
Matrix W:
[[-0.13724218   0.50303478]
 [ 0.24724326   0.16487119]
 [-0.02545159   0.24456476]
 [ 0.20694508  -0.11352904]
 [-0.15436582   0.28974518]
 [-0.39376952   0.05080104]
 [-0.41735106  -0.02287338]
 [ 0.30572896   0.09048885]
 [-0.30668347   0.00835233]
 [ 0.07554066   0.54977581]
 [-0.32613263  -0.20716433]
 [-0.36861022  -0.24902536]
 [-0.29669651   0.38022942]]
```

By executing the preceding code, we have created a 13 × 2-dimensional projection matrix, ***W***, from the top two eigenvectors.

Mirrored projections

Depending on which versions of NumPy and LAPACK you are using, you may obtain the matrix, $\boldsymbol{W}$, with its signs flipped. Please note that this is not an issue; if $\boldsymbol{v}$ is an eigenvector of a matrix, Σ, we have:

$$\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$$

Here, $\boldsymbol{v}$ is the eigenvector, and $-\boldsymbol{v}$ is also an eigenvector, which we can show as follows. Using basic algebra, we can multiply both sides of the equation by a scalar, α:

$$\alpha\Sigma\boldsymbol{v} = \alpha\lambda\boldsymbol{v}$$

Since matrix multiplication is associative for scalar multiplication, we can then rearrange this to the following:

$$\Sigma(\alpha\boldsymbol{v}) = \lambda(\alpha\boldsymbol{v})$$

Now, we can see that $\alpha\boldsymbol{v}$ is an eigenvector with the same eigenvalue, λ, for both $\alpha = 1$ and $\alpha = -1$. Hence, both $\boldsymbol{v}$ and $-\boldsymbol{v}$ are eigenvectors.

Using the projection matrix, we can now transform an example, $\boldsymbol{x}$ (represented as a 13-dimensional row vector), onto the PCA subspace (the principal components one and two) obtaining $\boldsymbol{x}'$, now a two-dimensional example vector consisting of two new features:

$$\boldsymbol{x}' = \boldsymbol{x}\boldsymbol{W}$$

```
>>> X_train_std[0].dot(w)
array([ 2.38299011,  0.45458499])
```

Similarly, we can transform the entire 124 × 13-dimensional training dataset onto the two principal components by calculating the matrix dot product:

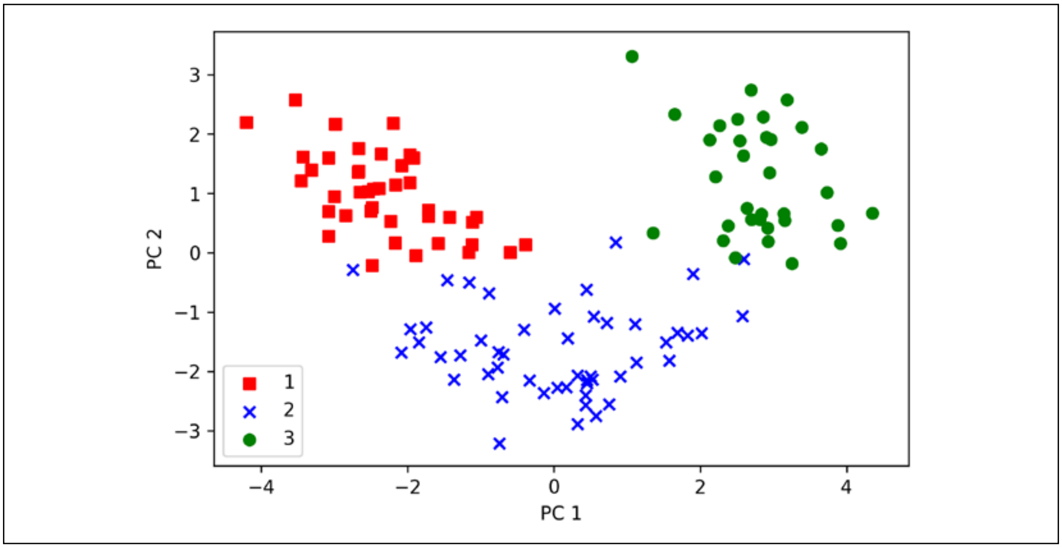
$$\boldsymbol{X}' = \boldsymbol{X}\boldsymbol{W}$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Lastly, let's visualize the transformed Wine training dataset, now stored as an 124 × 2 -dimensional matrix, in a two-dimensional scatterplot:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the data is more spread along the *x*-axis—the first principal component—than the second principal component (*y*-axis), which is consistent with the explained variance ratio plot that we created in the previous subsection. However, we can tell that a linear classifier will likely be able to separate the classes well:



Although we encoded the class label information for the purpose of illustration in the preceding scatter plot, we have to keep in mind that PCA is an unsupervised technique that doesn't use any class label information.

Principal component analysis in scikit-learn

Although the verbose approach in the previous subsection helped us to follow the inner workings of PCA, we will now discuss how to use the `PCA` class implemented in scikit-learn.

The PCA class is another one of scikit-learn's transformer classes, where we first fit the model using the training data before we transform both the training data and the test dataset using the same model parameters. Now, let's use the PCA class from scikit-learn on the Wine training dataset, classify the transformed examples via logistic regression, and visualize the decision regions via the plot_decision_regions function that we defined in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*:

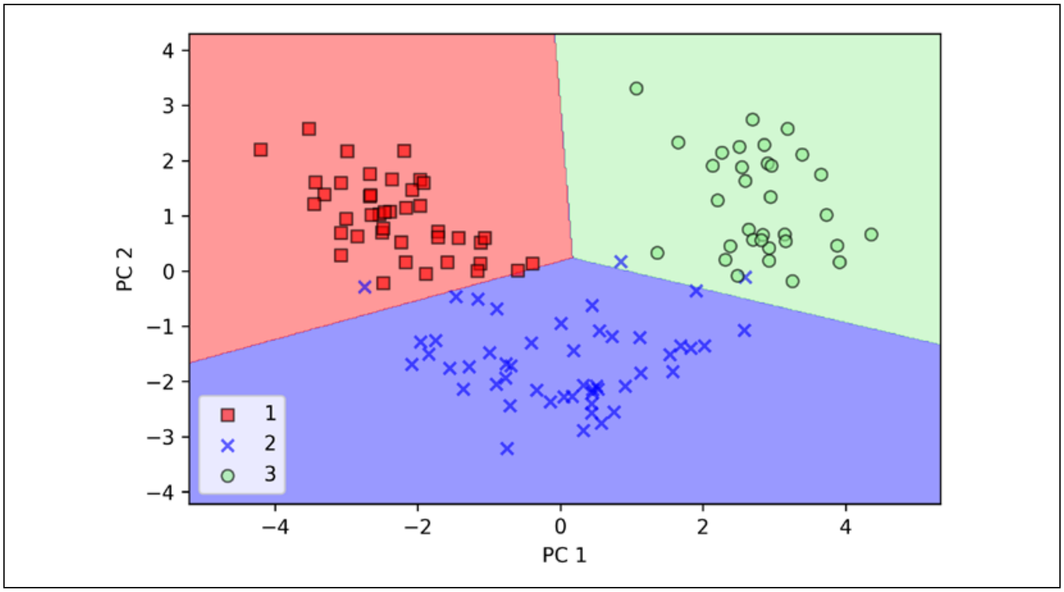
```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot examples by class
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.6,
                    color=cmap(idx),
                    edgecolor='black',
                    marker=markers[idx],
                    label=cl)
```

For your convenience, you can place the plot_decision_regions code shown above into a separate code file in your current working directory, for example, plot_decision_regions_script.py, and import it into your current Python session.

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> # initializing the PCA transformer and
>>> # logistic regression estimator:
```

```
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression(multi_class='ovr',
...                         random_state=1,
...                         solver='lbfgs')
>>> # dimensionality reduction:
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> # fitting the logistic regression model on the reduced dataset:
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

By executing the preceding code, we should now see the decision regions for the training data reduced to two principal component axes:

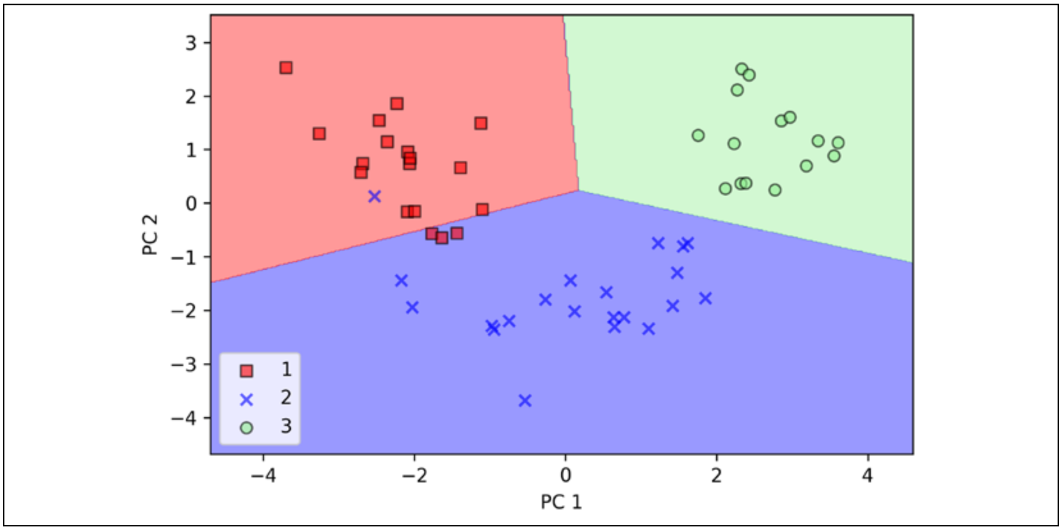


When we compare PCA projections via scikit-learn with our own PCA implementation, it can happen that the resulting plots are mirror images of each other. Note that this is not due to an error in either of those two implementations; the reason for this difference is that, depending on the eigensolver, eigenvectors can have either negative or positive signs.

Not that it matters, but we could simply revert the mirror image by multiplying the data by –1 if we wanted to; note that eigenvectors are typically scaled to unit length 1. For the sake of completeness, let's plot the decision regions of the logistic regression on the transformed test dataset to see if it can separate the classes well:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

After we plot the decision regions for the test dataset by executing the preceding code, we can see that logistic regression performs quite well on this small two-dimensional feature subspace and only misclassifies a few examples in the test dataset:



If we are interested in the explained variance ratios of the different principal components, we can simply initialize the `PCA` class with the `n_components` parameter set to `None`, so all principal components are kept and the explained variance ratio can then be accessed via the `explained_variance_ratio_` attribute:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.36951469, 0.18434927, 0.11815159, 0.07334252,
        0.06422108, 0.05051724, 0.03954654, 0.02643918,
        0.02389319, 0.01629614, 0.01380021, 0.01172226,
        0.00820609])
```

Note that we set `n_components=None` when we initialized the `PCA` class so that it will return all principal components in a sorted order, instead of performing a dimensionality reduction.

Supervised data compression via linear discriminant analysis

LDA can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of overfitting due to the curse of dimensionality in non-regularized models. The general concept behind LDA is very similar to PCA, but whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset, the goal in LDA is to find the feature subspace that optimizes class separability. In the following sections, we will discuss the similarities between LDA and PCA in more detail and walk through the LDA approach step by step.

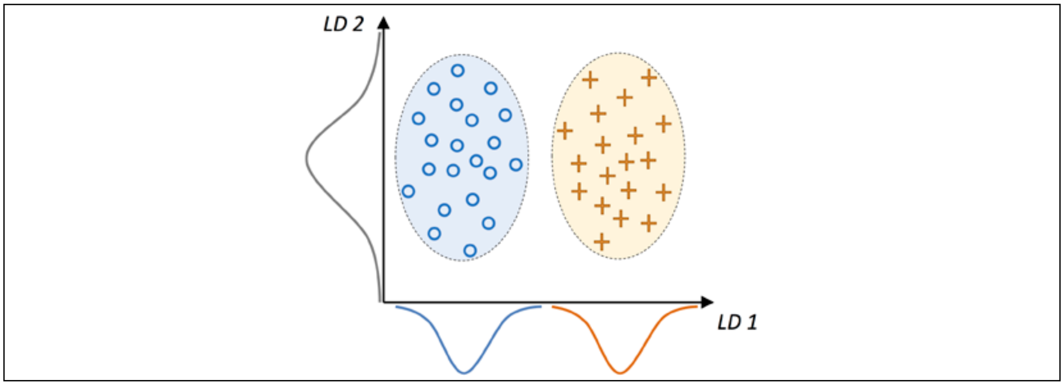
Principal component analysis versus linear discriminant analysis

Both PCA and LDA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez reported that preprocessing via PCA tends to result in better classification results in an image recognition task in certain cases, for instance, if each class consists of only a small number of examples (*PCA Versus LDA*, *A. M. Martinez* and *A. C. Kak*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(2): 228-233, *2001*).

Fisher LDA

LDA is sometimes also called Fisher's LDA. Ronald A. Fisher initially formulated *Fisher's Linear Discriminant* for two-class classification problems in 1936 (*The Use of Multiple Measurements in Taxonomic Problems*, *R. A. Fisher*, *Annals of Eugenics*, 7(2): 179-188, *1936*). Fisher's linear discriminant was later generalized for multi-class problems by C. Radhakrishna Rao under the assumption of equal class covariances and normally distributed classes in 1948, which we now call LDA (*The Utilization of Multiple Measurements in Problems of Biological Classification*, *C. R. Rao*, *Journal of the Royal Statistical Society*. Series B (Methodological), 10(2): 159-203, *1948*).

The following figure summarizes the concept of LDA for a two-class problem. Examples from class 1 are shown as circles, and examples from class 2 are shown as crosses:



A linear discriminant, as shown on the *x*-axis (LD 1), would separate the two normal distributed classes well. Although the exemplary linear discriminant shown on the *y*-axis (LD 2) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the training examples are statistically independent of each other. However, even if one, or more, of those assumptions is (slightly) violated, LDA for dimensionality reduction can still work reasonably well (*Pattern Classification 2nd Edition*, *R. O. Duda*, *P. E. Hart*, and *D. G. Stork*, *New York*, *2001*).

The inner workings of linear discriminant analysis

Before we dive into the code implementation, let's briefly summarize the main steps that are required to perform LDA:

1. Standardize the *d*-dimensional dataset (*d* is the number of features).
2. For each class, compute the *d*-dimensional mean vector.
3. Construct the between-class scatter matrix, $\boldsymbol{S}_B$, and the within-class scatter matrix, $\boldsymbol{S}_w$.

4. Compute the eigenvectors and corresponding eigenvalues of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$.
5. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
6. Choose the *k* eigenvectors that correspond to the *k* largest eigenvalues to construct a $d \times k$-dimensional transformation matrix, $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
7. Project the examples onto the new feature subspace using the transformation matrix, $\boldsymbol{W}$.

As we can see, LDA is quite similar to PCA in the sense that we are decomposing matrices into eigenvalues and eigenvectors, which will form the new lower-dimensional feature space. However, as mentioned before, LDA takes class label information into account, which is represented in the form of the mean vectors computed in step 2. In the following sections, we will discuss these seven steps in more detail, accompanied by illustrative code implementations.

Computing the scatter matrices

Since we already standardized the features of the Wine dataset in the PCA section at the beginning of this chapter, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector, $\boldsymbol{m}_i$, stores the mean feature value, μ_m, with respect to the examples of class *i*:

$$\boldsymbol{m}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i} \boldsymbol{x}_m$$

This results in three mean vectors:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1, 2, 3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                 X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' %(label, mean_vecs[label-1]))
MV 1: [ 0.9066  -0.3497  0.3201  -0.7189  0.5056  0.8807  0.9589
-0.5516
0.5416  0.2338  0.5897  0.6563  1.2075]
MV 2: [-0.8749  -0.2848  -0.3735  0.3157  -0.3848  -0.0433  0.0635
-0.0946
0.0703  -0.8286  0.3144  0.3608  -0.7253]
MV 3: [ 0.1992  0.866  0.1682  0.4148  -0.0451  -1.0286  -1.2876
0.8287
-0.7795  0.9649  -1.209  -1.3622  -0.4013]
```

Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S}_W$:

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$

This is calculated by summing up the individual scatter matrices, $\boldsymbol{S}_i$, of each individual class *i*:

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
>>> for row in X_train_std[y_train == label]:
...     row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...     class_scatter += (row - mv).dot((row - mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: %sx%s' % (
...       S_W.shape[0], S_W.shape[1]))
Within-class scatter matrix: 13x13
```

The assumption that we are making when we are computing the scatter matrices is that the class labels in the training dataset are uniformly distributed. However, if we print the number of class labels, we see that this assumption is violated:

```
>>> print('Class label distribution: %s'
...       % np.bincount(y_train)[1:])
Class label distribution: [41 50 33]
```

Thus, we want to scale the individual scatter matrices, $\boldsymbol{S}_i$, before we sum them up as scatter matrix $\boldsymbol{S}_W$. When we divide the scatter matrices by the number of class-examples, n_i, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix, Σ_i—the covariance matrix is a normalized version of the scatter matrix:

$$\Sigma_i = \frac{1}{n_i}\boldsymbol{S}_i = \frac{1}{n_i}\sum_{\boldsymbol{x}\in D_i}(\boldsymbol{x}-\boldsymbol{m}_i)(\boldsymbol{x}-\boldsymbol{m}_i)^T$$

The code for computing the scaled within-class scatter matrix is as follows:

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: %sx%s'
...      % (S_W.shape[0], S_W.shape[1]))
Scaled within-class scatter matrix: 13x13
```

After we compute the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\boldsymbol{S_B}$:

$$\boldsymbol{S_B} = \sum_{i=1}^{\boldsymbol{c}} n_i(\boldsymbol{m}_i - \boldsymbol{m})(\boldsymbol{m}_i - \boldsymbol{m})^T$$

Here, $\boldsymbol{m}$ is the overall mean that is computed, including examples from all $\boldsymbol{c}$ classes:

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...     n = X_train_std[y_train == i + 1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1) # make column vector
...     mean_overall = mean_overall.reshape(d, 1)
...     S_B += n * (mean_vec - mean_overall).dot(
...     (mean_vec - mean_overall).T)
>>> print('Between-class scatter matrix: %sx%s' % (
...                S_B.shape[0], S_B.shape[1]))
Between-class scatter matrix: 13x13
```

Selecting linear discriminants for the new feature subspace

The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$:

```
>>> eigen_vals, eigen_vecs =\
...     np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

After we compute the eigenpairs, we can sort the eigenvalues in descending order:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...               key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in descending order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])


Eigenvalues in descending order:

349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

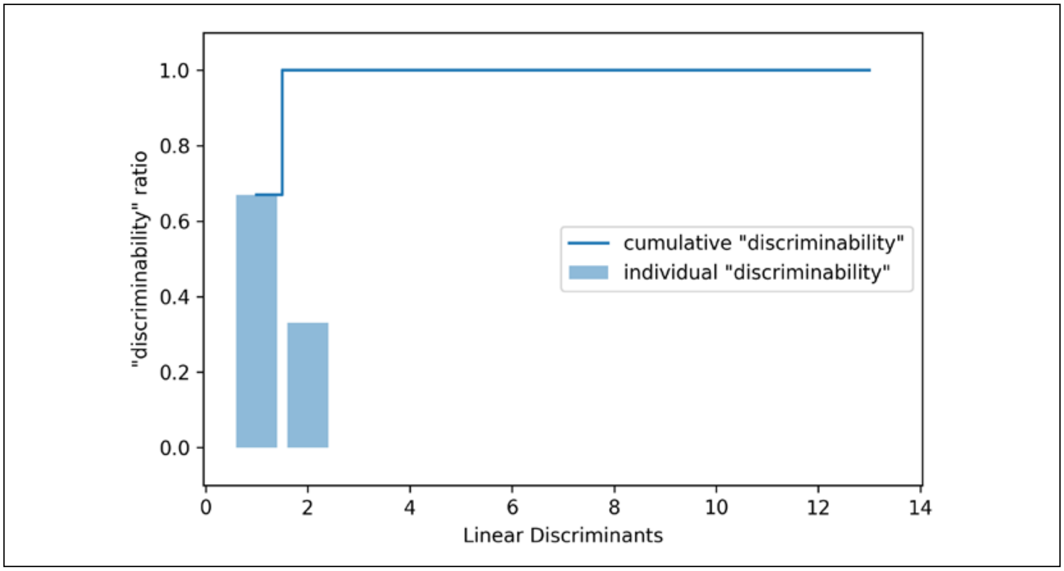
In LDA, the number of linear discriminants is at most *c*−1, where *c* is the number of class labels, since the in-between scatter matrix, $\boldsymbol{S_B}$, is the sum of *c* matrices with rank one or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating-point arithmetic in NumPy).

> **Collinearity**
>
> Note that in the rare case of perfect collinearity (all aligned example points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues, similar to the explained variance plot that we created in the PCA section. For simplicity, we will call the content of class-discriminatory information **discriminability**:

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...         label='Individual "discriminability"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='Cumulative "discriminability"')
>>> plt.ylabel('"Discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting figure, the first two linear discriminants alone capture 100 percent of the useful information in the Wine training dataset:



Let's now stack the two most discriminative eigenvector columns to create the transformation matrix, ***W***:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
 [[-0.1481  -0.4092]
  [ 0.0908  -0.1577]
  [-0.0168  -0.3537]
  [ 0.1484   0.3223]
  [-0.0163  -0.0817]
  [ 0.1913   0.0842]
  [-0.7338   0.2823]
  [-0.075   -0.0102]
  [ 0.0018   0.0907]
  [ 0.294   -0.2152]
  [-0.0328   0.2747]
  [-0.3547  -0.0124]
  [-0.3915  -0.5958]]
```

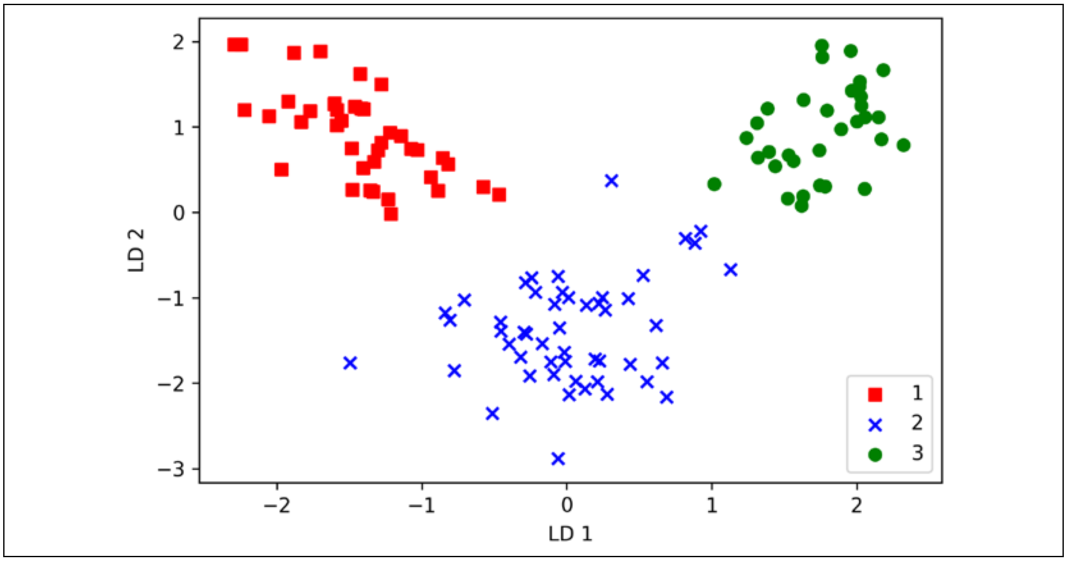
Projecting examples onto the new feature space

Using the transformation matrix, ***W***, that we created in the previous subsection, we can now transform the training dataset by multiplying the matrices:

$$\boldsymbol{X}' = \boldsymbol{X}\boldsymbol{W}$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the three Wine classes are now perfectly linearly separable in the new feature subspace:



LDA via scikit-learn

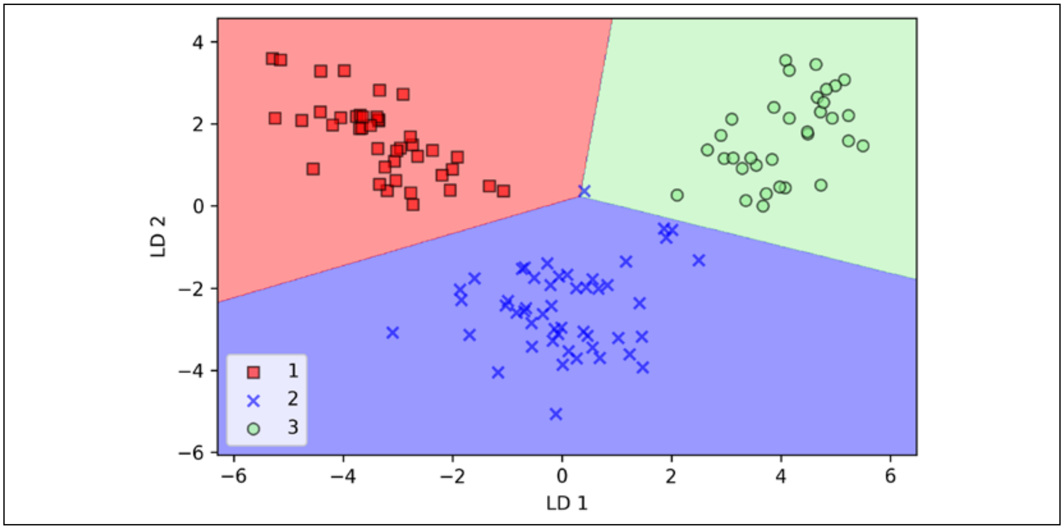
That step-by-step implementation was a good exercise to understand the inner workings of an LDA and understand the differences between LDA and PCA. Now, let's look at the `LDA` class implemented in scikit-learn:

```
>>> # the following import statement is one line
>>> from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Next, let's see how the logistic regression classifier handles the lower-dimensional training dataset after the LDA transformation:

```
>>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                         solver='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

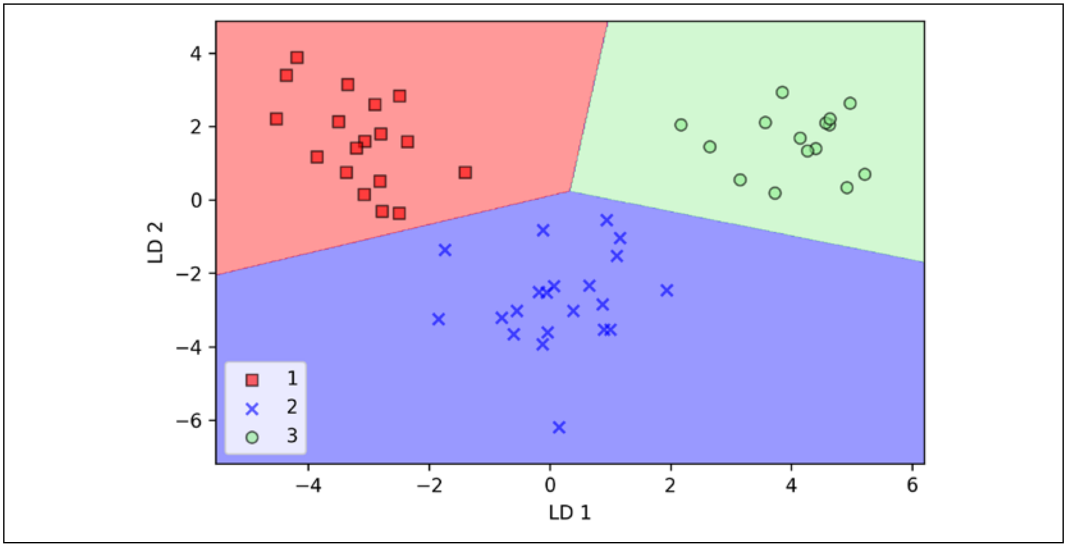
Looking at the resulting plot, we can see that the logistic regression model misclassifies one of the examples from class 2:



By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression model classifies all examples in the training dataset correctly. However, and more importantly, let's take a look at the results on the test dataset:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the following plot, the logistic regression classifier is able to get a perfect accuracy score for classifying the examples in the test dataset by only using a two-dimensional feature subspace, instead of the original 13 Wine features:

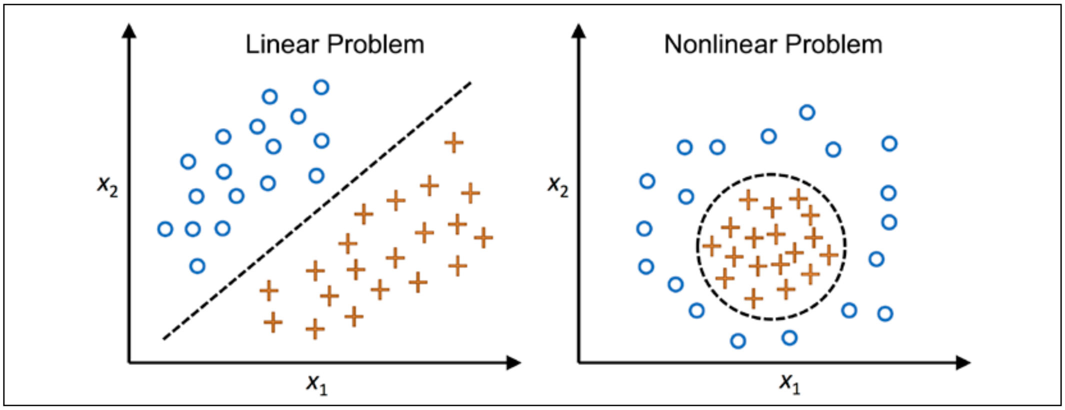


Using kernel principal component analysis for nonlinear mappings

Many machine learning algorithms make assumptions about the linear separability of the input data. You have learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few.

However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice.

In this section, we will take a look at a kernelized version of PCA, or KPCA, which relates to the concepts of kernel SVM that you will remember from *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*. Using KPCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.



Kernel functions and the kernel trick

As you will remember from our discussion about kernel SVMs in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we can tackle nonlinear problems by projecting them onto a new feature space of higher dimensionality where the classes become linearly separable. To transform the examples $\boldsymbol{x} \in \mathbb{R}^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function, ϕ:

$$\phi: \quad \mathbb{R}^d \longrightarrow \mathbb{R}^k \qquad (k \gg d)$$

We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space.

For example, if we had a feature vector $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ is a column vector consisting of $\boldsymbol{d}$ features) with two dimensions (d = 2), a potential mapping onto a 3D-space could be:

$$\boldsymbol{x} = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$\boldsymbol{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$$

In other words, we perform a nonlinear mapping via KPCA that transforms the data onto a higher-dimensional space. We then use standard PCA in this higher-dimensional space to project the data back onto a lower-dimensional space where the examples can be separated by a linear classifier (under the condition that the examples can be separated by density in the input space). However, one downside of this approach is that it is computationally very expensive, and this is where we use the **kernel trick**. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Before we proceed with more details about the kernel trick to tackle this computationally expensive problem, let's think back to the standard PCA approach that we implemented at the beginning of this chapter. We computed the covariance between two features, k and j, as follows:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Since the standardizing of features centers them at mean zero, for instance, $\mu_j = 0$ and $\mu_k = 0$, we can simplify this equation as follows:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance matrix, Σ:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}^{(i)} \, \boldsymbol{x}^{(i)^T}$$

Bernhard Scholkopf generalized this approach (*Kernel principal component analysis*, *B. Scholkopf*, *A. Smola*, and *K.R. Muller*, pages 583-588, *1997*) so that we can replace the dot products between examples in the original feature space with the nonlinear feature combinations via ϕ:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T$$

To obtain the eigenvectors—the principal components—from this covariance matrix, we have to solve the following equation:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T \boldsymbol{v} = \lambda \boldsymbol{v}$$

$$\Rightarrow v = \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^T \boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{a}^{(\boldsymbol{i})}\phi\left(\boldsymbol{x}^{(i)}\right)$$

Here, λ and $\boldsymbol{v}$ are the eigenvalues and eigenvectors of the covariance matrix, Σ, and $\boldsymbol{a}$ can be obtained by extracting the eigenvectors of the kernel (similarity) matrix, $\boldsymbol{K}$, as you will see in the following paragraphs.

Deriving the kernel matrix

The derivation of the kernel matrix can be shown as follows. First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$-dimensional matrix:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi\left(\boldsymbol{x}^{(i)}\right)\phi\left(\boldsymbol{x}^{(i)}\right)^{T} = \frac{1}{n}\,\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})$$

Now, we can write the eigenvector equation as follows:

$$v = \frac{1}{n}\sum_{i=1}^{n}a^{(i)}\phi\left(\boldsymbol{x}^{(i)}\right) = \lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

Since $\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$, we get:

$$\frac{1}{n}\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a} = \lambda\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

Multiplying it by $\phi(\boldsymbol{X})$ on both sides yields the following result:

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a} = \lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}\boldsymbol{a} = \lambda\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}$$

Here, $\boldsymbol{K}$ is the similarity (kernel) matrix:

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^{T}$$

As you might recall from the *Solving nonlinear problems using a kernel SVM* section in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the examples, $\boldsymbol{x}$, under ϕ explicitly by using a kernel function, κ, so that we don't need to calculate the eigenvectors explicitly:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^{T}\phi\left(\boldsymbol{x}^{(j)}\right)$$

In other words, what we obtain after KPCA are the examples already projected onto the respective components, rather than constructing a transformation matrix as in the standard PCA approach. Basically, the kernel function (or simply kernel) can be understood as a function that calculates a dot product between two vectors—a measure of similarity.

The most commonly used kernels are as follows:

- The polynomial kernel:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \left(\boldsymbol{x}^{(i)^T}\boldsymbol{x}^{(j)} + \theta\right)^p$$

  Here, θ is the threshold and p is the power that has to be specified by the user.

- The hyperbolic tangent (sigmoid) kernel:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \tanh\left(\eta\boldsymbol{x}^{(i)^T}\boldsymbol{x}^{(j)} + \theta\right)$$

- The **radial basis function** (**RBF**) or Gaussian kernel, which we will use in the following examples in the next subsection:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right)$$

  It is often written in the following form, introducing the variable $\gamma = \frac{1}{2\sigma^2}$.

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

To summarize what we have learned so far, we can define the following three steps to implement an RBF KPCA:

1. We compute the kernel (similarity) matrix, $\boldsymbol{K}$, where we need to calculate the following:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

We do this for each pair of examples:

$$\boldsymbol{K} = \begin{bmatrix} \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(n)}\right) \\ \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right) & \left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(2)}, \boldsymbol{x}^{(n)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(1)}\right) & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(2)}\right) & \cdots & \kappa\left(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(n)}\right) \end{bmatrix}$$

For example, if our dataset contains 100 training examples, the symmetric kernel matrix of the pairwise similarities would be 100 × 100-dimensional.

2. We center the kernel matrix, $\boldsymbol{K}$, using the following equation:

   $$\boldsymbol{K}' = \boldsymbol{K} - \boldsymbol{1}_{\boldsymbol{n}}\boldsymbol{K} - \boldsymbol{K}\boldsymbol{1}_{\boldsymbol{n}} + \boldsymbol{1}_{\boldsymbol{n}}\boldsymbol{K}\boldsymbol{1}_{\boldsymbol{n}}$$

   Here, $\boldsymbol{1}_{\boldsymbol{n}}$ is an $n \times n$-dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

3. We collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, which are ranked by decreasing magnitude. In contrast to standard PCA, the eigenvectors are not the principal component axes, but the examples already projected onto these axes.

At this point, you may be wondering why we need to center the kernel matrix in the second step. We previously assumed that we are working with standardized data, where all features have mean zero when we formulate the covariance matrix and replace the dot-products with the nonlinear feature combinations via ϕ. Thus, the centering of the kernel matrix in the second step becomes necessary, since we do not compute the new feature space explicitly so that we cannot guarantee that the new feature space is also centered at zero.

In the next section, we will put those three steps into action by implementing a KPCA in Python.

Implementing a kernel principal component analysis in Python

In the previous subsection, we discussed the core concepts behind KPCA. Now, we are going to implement an RBF KPCA in Python following the three steps that summarized the KPCA approach. Using some SciPy and NumPy helper functions, we will see that implementing a KPCA is actually really simple:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_examples, n_features]

    gamma: float
        Tuning parameter of the RBF kernel

    n_components: int
        Number of principal components to return

    Returns
    ------------
    X_pc: {NumPy ndarray}, shape = [n_examples, k_features]
        Projected dataset

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)

    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # scipy.linalg.eigh returns them in ascending order
    eigvals, eigvecs = eigh(K)
    eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]
    # Collect the top k eigenvectors (projected examples)
    X_pc = np.column_stack([eigvecs[:, i]
                           for i in range(n_components)])

    return X_pc
```

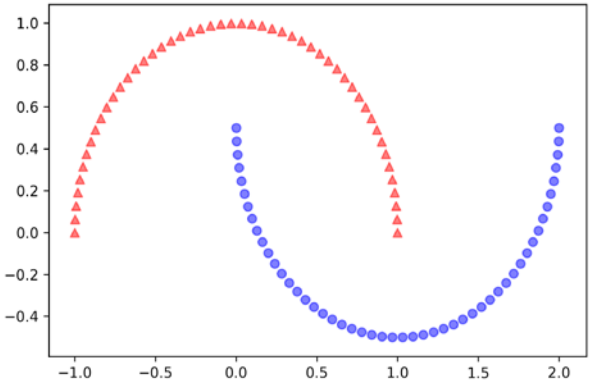
One downside of using an RBF KPCA for dimensionality reduction is that we have to specify the γ parameter a priori. Finding an appropriate value for γ requires experimentation and is best done using algorithms for parameter tuning, for example, performing a grid search, which we will discuss in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Example 1 – separating half-moon shapes

Now, let us apply our `rbf_kernel_pca` on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 example points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

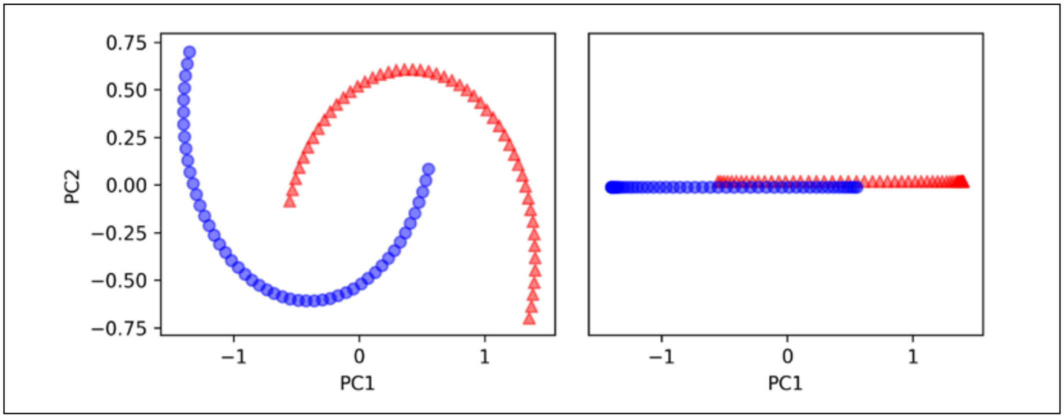
For the purposes of illustration, the half-moon of triangle symbols will represent one class, and the half-moon depicted by the circle symbols will represent the examples from another class:



Clearly, these two half-moon shapes are not linearly separable, and our goal is to *unfold* the half-moons via KPCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see how the dataset looks if we project it onto the principal components via standard PCA:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Clearly, we can see in the resulting figure that a linear classifier would be unable to perform well on the dataset transformed via standard PCA:



Note that when we plotted the first principal component only (right subplot), we shifted the triangular examples slightly upward and the circular examples slightly downward to better visualize the class overlap. As the left subplot shows, the original half-moon shapes are only slightly sheared and flipped across the vertical center—this transformation would not help a linear classifier in discriminating between circles and triangles. Similarly, the circles and triangles corresponding to the two half-moon shapes are not linearly separable if we project the dataset onto a one-dimensional feature axis, as shown in the right subplot.

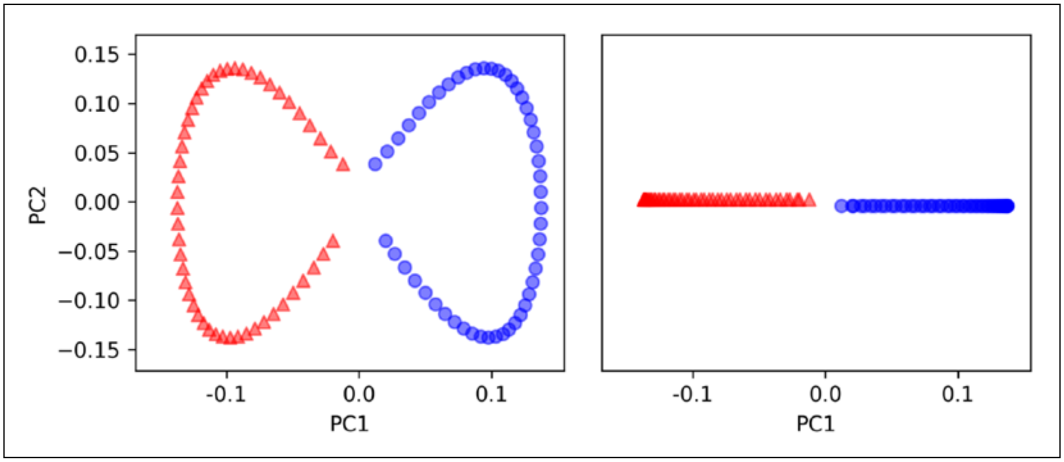
PCA versus LDA

Please remember that PCA is an unsupervised method and does not use class label information in order to maximize the variance in contrast to LDA. Here, the triangle and circle symbols were just added for visualization purposes to indicate the degree of separation.

Now, let's try out our kernel PCA function, `rbf_kernel_pca`, which we implemented in the previous subsection:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

We can now see that the two classes (circles and triangles) are linearly well separated so that we have a suitable training dataset for linear classifiers:



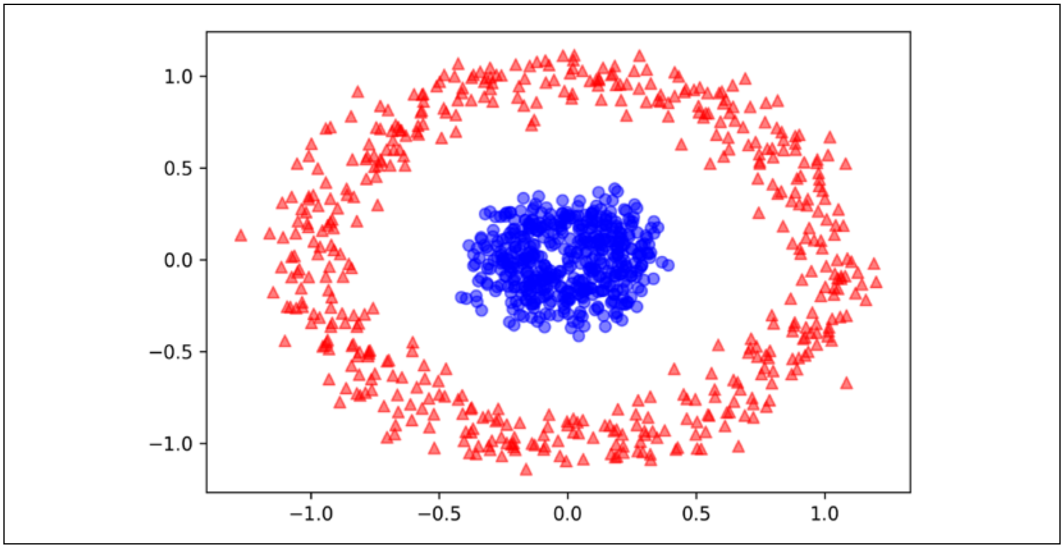
Unfortunately, there is no universal value for the tuning parameter, γ, that works well for different datasets. Finding a γ value that is appropriate for a given problem requires experimentation. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss techniques that can help us to automate the task of optimizing such tuning parameters. Here, we will use values for γ that have been found to produce good results.

Example 2 – separating concentric circles

In the previous subsection, we saw how to separate half-moon shapes via KPCA. Since we put so much effort into understanding the concepts of KPCA, let's take a look at another interesting example of a nonlinear problem, concentric circles:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                     random_state=123, noise=0.1,
...                     factor=0.2)
>>> plt.scatter(X[y == 0, 0], X[y == 0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y == 1, 0], X[y == 1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

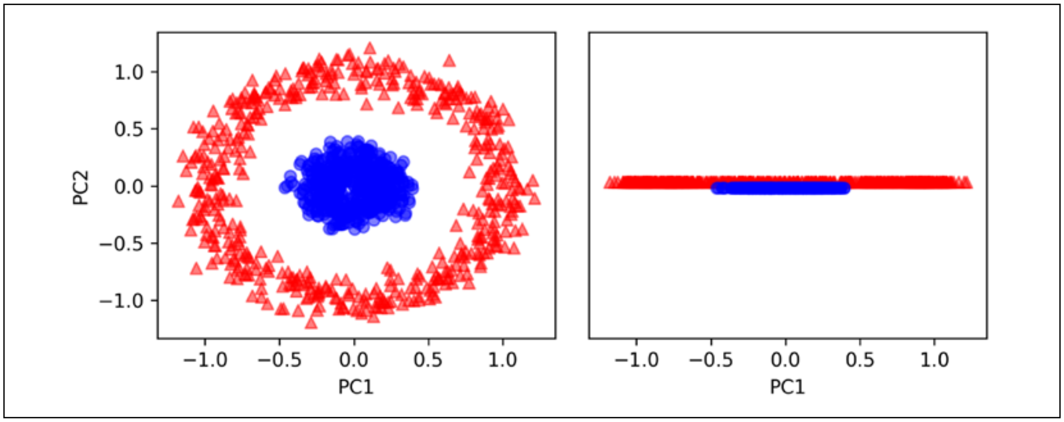
Again, we assume a two-class problem where the triangle shapes represent one class, and the circle shapes represent another class:



Let's start with the standard PCA approach to compare it to the results of the RBF kernel PCA:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

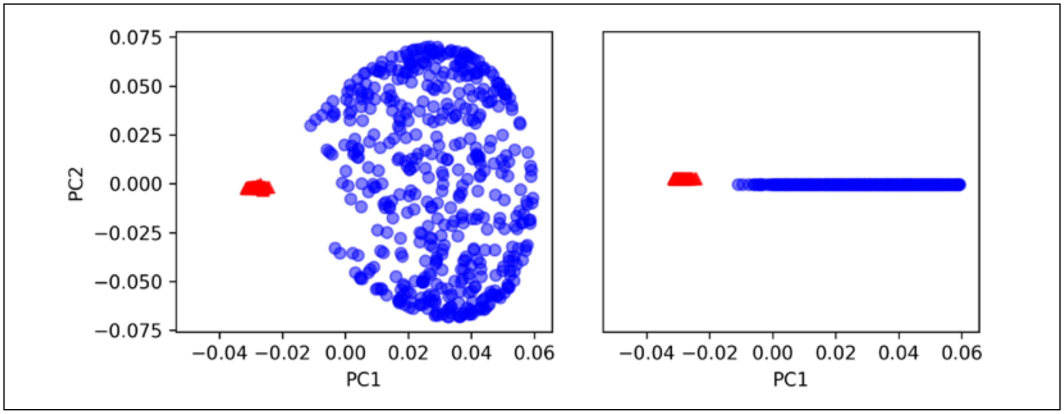
Again, we can see that standard PCA is not able to produce results suitable for training a linear classifier:



Given an appropriate value for γ, let's see if we are luckier using the RBF KPCA implementation:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Again, the RBF KPCA projected the data onto a new subspace where the two classes become linearly separable:



Projecting new data points

In the two previous example applications of KPCA, the half-moon shapes and the concentric circles, we projected a single dataset onto a new feature. In real applications, however, we may have more than one dataset that we want to transform, for example, training and test data, and typically also new examples we will collect after the model building and evaluation. In this section, you will learn how to project data points that were not part of the training dataset.

As you will remember from the standard PCA approach at the beginning of this chapter, we project data by calculating the dot product between a transformation matrix and the input examples; the columns of the projection matrix are the top k eigenvectors ($\boldsymbol{v}$) that we obtained from the covariance matrix.

Now, the question is how we can transfer this concept to KPCA. If we think back to the idea behind KPCA, we will remember that we obtained an eigenvector ($\boldsymbol{a}$) of the centered kernel matrix (not the covariance matrix), which means that those are the examples that are already projected onto the principal component axis, $\boldsymbol{v}$. Thus, if we want to project a new example, $\boldsymbol{x}'$, onto this principal component axis, we will need to compute the following:

$$\phi(\boldsymbol{x}')^T\boldsymbol{v}$$

Fortunately, we can use the kernel trick so that we don't have to calculate the projection, $\phi(\boldsymbol{x}')^T\boldsymbol{v}$, explicitly. However, it is worth noting that KPCA, in contrast to standard PCA, is a memory-based method, which means that we have to *reuse the original training dataset each time to project new examples*.

We have to calculate the pairwise RBF kernel (similarity) between each ith example in the training dataset and the new example, $\boldsymbol{x}'$:

$$\begin{aligned}\phi(\boldsymbol{x}')^T\boldsymbol{v} &= \sum_i a^{(i)}\phi(\boldsymbol{x}')^{\boldsymbol{T}}\phi\left(\boldsymbol{x}^{(i)}\right)\\ &= \sum_i a^{(i)}\kappa\left(\boldsymbol{x}', \boldsymbol{x}^{(i)}\right)\end{aligned}$$

Here, the eigenvectors, $\boldsymbol{a}$, and eigenvalues, λ, of the kernel matrix, $\boldsymbol{K}$, satisfy the following condition in the equation:

$$\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}$$

After calculating the similarity between the new examples and the examples in the training dataset, we have to normalize the eigenvector, $\boldsymbol{a}$, by its eigenvalue. Thus, let's modify the `rbf_kernel_pca` function that we implemented earlier so that it also returns the eigenvalues of the kernel matrix:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.

    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_examples, n_features]

    gamma: float
        Tuning parameter of the RBF kernel

    n_components: int
        Number of principal components to return

    Returns
    ------------
    alphas {NumPy ndarray}, shape = [n_examples, k_features]
        Projected dataset

    lambdas: list
```

```
         Eigenvalues

    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')

    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)

    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)

    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Obtaining eigenpairs from the centered kernel matrix
    # scipy.linalg.eigh returns them in ascending order
    eigvals, eigvecs = eigh(K)
    eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]

    # Collect the top k eigenvectors (projected examples)
    alphas = np.column_stack([eigvecs[:, i]
                             for i in range(n_components)])

    # Collect the corresponding eigenvalues
    lambdas = [eigvals[i] for i in range(n_components)]
    return alphas, lambdas
```

Now, let's create a new half-moon dataset and project it onto a one-dimensional subspace using the updated RBF KPCA implementation:

```
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas = rbf_kernel_pca(X, gamma=15, n_components=1)
```

To make sure that we implemented the code for projecting new examples, let's assume that the 26th point from the half-moon dataset is a new data point, $\boldsymbol{x}'$, and our task is to project it onto this new subspace:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25] # original projection
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...                 (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
...     return k.dot(alphas / lambdas)
```

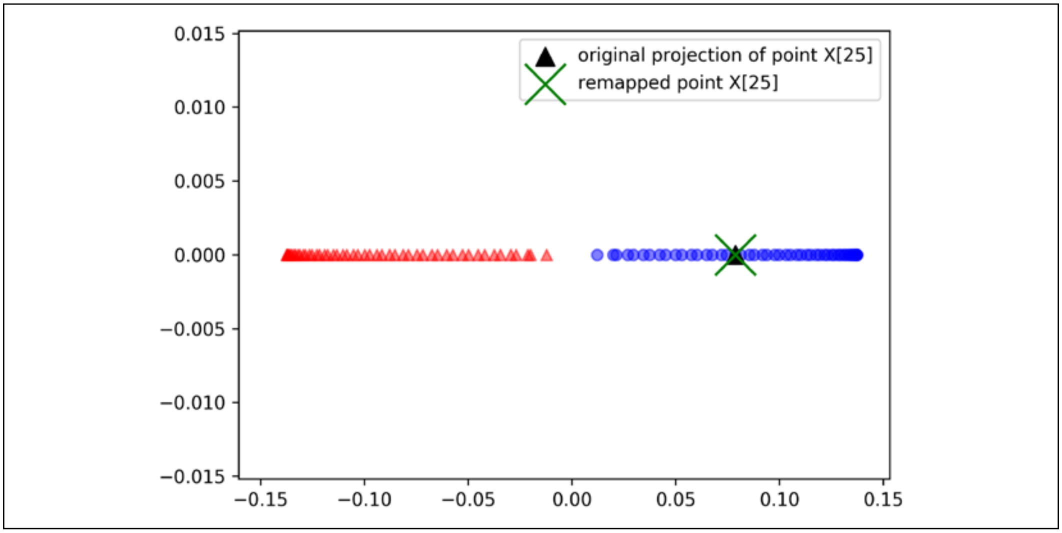
By executing the following code, we are able to reproduce the original projection. Using the `project_x` function, we will be able to project any new data example as well. The code is as follows:

```
>>> x_reproj = project_x(x_new, X,
...            gamma=15, alphas=alphas,
...            lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Lastly, let's visualize the projection on the first principal component:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...             color='red', marker='^',alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...             color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...             label='Original projection of point X[25]',
...             marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...             label='Remapped point X[25]',
...             marker='x', s=500)
>>> plt.yticks([], [])
>>> plt.legend(scatterpoints=1)
>>> plt.tight_layout()
>>> plt.show()
```

As we can now also see in the following scatterplot, we mapped the example, $\boldsymbol{x}'$, onto the first principal component correctly:



Kernel principal component analysis in scikit-learn

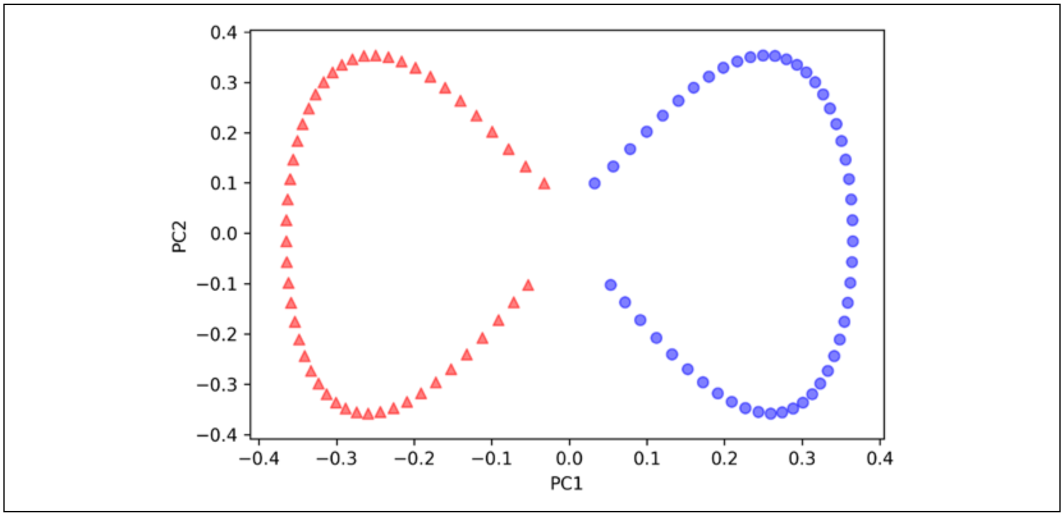
For our convenience, scikit-learn implements a KPCA class in the `sklearn.decomposition` submodule. The usage is similar to the standard PCA class, and we can specify the kernel via the `kernel` parameter:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                 kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

To check that we get results that are consistent with our own KPCA implementation, let's plot the transformed half-moon shape data onto the first two principal components:

```
>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see, the results of scikit-learn's `KernelPCA` are consistent with our own implementation:



Manifold learning

The scikit-learn library also implements advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. The interested reader can find a nice overview of the current implementations in scikit-learn, complemented by illustrative examples, at `http://scikit-learn.org/stable/modules/manifold.html`.

Summary

In this chapter, you learned about three different, fundamental dimensionality reduction techniques for feature extraction: standard PCA, LDA, and KPCA. Using PCA, we projected data onto a lower-dimensional subspace to maximize the variance along the orthogonal feature axes, while ignoring the class labels. LDA, in contrast to PCA, is a technique for supervised dimensionality reduction, which means that it considers class information in the training dataset to attempt to maximize the class-separability in a linear feature space.

Lastly, you learned about a nonlinear feature extractor, KPCA. Using the kernel trick and a temporary projection into a higher-dimensional feature space, you were ultimately able to compress datasets consisting of nonlinear features onto a lower-dimensional subspace where the classes became linearly separable.

Equipped with these essential preprocessing techniques, you are now well prepared to learn about the best practices for efficiently incorporating different preprocessing techniques and evaluating the performance of different models in the next chapter.

6

Learning Best Practices for Model Evaluation and Hyperparameter Tuning

In the previous chapters, you learned about the essential machine learning algorithms for classification and how to get our data into shape before we feed it into those algorithms. Now, it's time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the performance of the models. In this chapter, we will learn how to do the following:

- Assess the performance of machine learning models
- Diagnose the common problems of machine learning algorithms
- Fine-tune machine learning models
- Evaluate predictive models using different performance metrics

Streamlining workflows with pipelines

When we applied different preprocessing techniques in the previous chapters, such as standardization for feature scaling in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, or principal component analysis for data compression in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned that we have to reuse the parameters that were obtained during the fitting of the training data to scale and compress any new data, such as the examples in the separate test dataset. In this section, you will learn about an extremely handy tool, the `Pipeline` class in scikit-learn. It allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Loading the Breast Cancer Wisconsin dataset

In this chapter, we will be working with the Breast Cancer Wisconsin dataset, which contains 569 examples of malignant and benign tumor cells. The first two columns in the dataset store the unique ID numbers of the examples and the corresponding diagnoses (`M` = malignant, `B` = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited in the UCI Machine Learning Repository, and more detailed information about this dataset can be found at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

Obtaining the Breast Cancer Wisconsin dataset

You can find a copy of the dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data` is temporarily unavailable. For instance, to load the dataset from a local directory, you can replace the following lines:

```
df = pd.read_csv(
     'https://archive.ics.uci.edu/ml/'
     'machine-learning-databases'
     '/breast-cancer-wisconsin/wdbc.data',
     header=None)
```

with these:

```
df = pd.read_csv(
     'your/local/path/to/wdbc.data',
     header=None)
```

In this section, we will read in the dataset and split it into training and test datasets in three simple steps:

1. We will start by reading in the dataset directly from the UCI website using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                  'machine-learning-databases'
...                  '/breast-cancer-wisconsin/wdbc.data',
...                  header=None)
```

2. Next, we will assign the 30 features to a NumPy array, `X`. Using a `LabelEncoder` object, we will transform the class labels from their original string representation (`'M'` and `'B'`) into integers:

```
>>> from sklearn.preprocessing import LabelEncoder

>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> le.classes_
array(['B', 'M'], dtype=object)
```

   After encoding the class labels (diagnosis) in an array, `y`, the malignant tumors are now represented as class `1`, and the benign tumors are represented as class `0`, respectively. We can double-check this mapping by calling the `transform` method of the fitted `LabelEncoder` on two dummy class labels:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. Before we construct our first model pipeline in the following subsection, let's divide the dataset into a separate training dataset (80 percent of the data) and a separate test dataset (20 percent of the data):

```
>>> from sklearn.model_selection import train_test_split

>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                      test_size=0.20,
...                      stratify=y,
...                      random_state=1)
```

Combining transformers and estimators in a pipeline

In the previous chapter, you learned that many learning algorithms require input features on the same scale for optimal performance. Since the features in the Breast Cancer Wisconsin dataset are measured on various different scales, we will standardize the columns in the Breast Cancer Wisconsin dataset before we feed them to a linear classifier, such as logistic regression. Furthermore, let's assume that we want to compress our data from the initial 30 dimensions onto a lower two-dimensional subspace via **principal component analysis** (**PCA**), a feature extraction technique for dimensionality reduction that was introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Instead of going through the model fitting and data transformation steps for the training and test datasets separately, we can chain the `StandardScaler`, `PCA`, and `LogisticRegression` objects in a pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression(random_state=1,
...                                            solver='lbfgs'))
>>> pipe_lr.fit(X_train, y_train)
>>> y_pred = pipe_lr.predict(X_test)
>>> print('Test Accuracy: %.3f' % pipe_lr.score(X_test, y_test))
Test Accuracy: 0.956
```

The `make_pipeline` function takes an arbitrary number of scikit-learn transformers (objects that support the `fit` and `transform` methods as input), followed by a scikit-learn estimator that implements the `fit` and `predict` methods. In our preceding code example, we provided two transformers, `StandardScaler` and `PCA`, and a `LogisticRegression` estimator as inputs to the `make_pipeline` function, which constructs a scikit-learn `Pipeline` object from these objects.

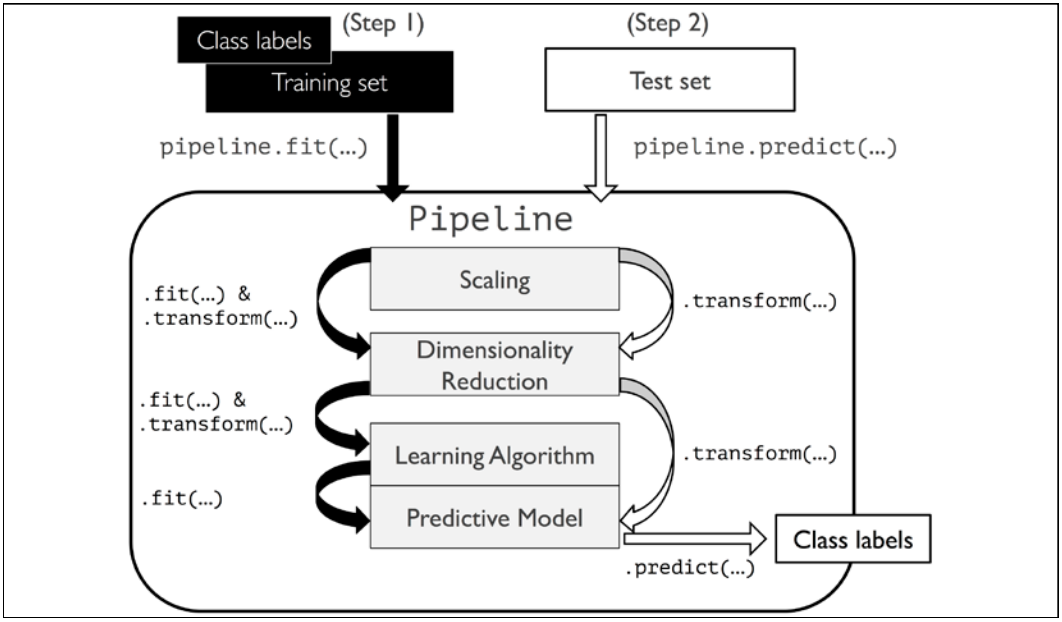
We can think of a scikit-learn `Pipeline` as a meta-estimator or wrapper around those individual transformers and estimators. If we call the `fit` method of `Pipeline`, the data will be passed down a series of transformers via `fit` and `transform` calls on these intermediate steps until it arrives at the estimator object (the final element in a pipeline). The estimator will then be fitted to the transformed training data.

When we executed the `fit` method on the `pipe_lr` pipeline in the preceding code example, `StandardScaler` first performed `fit` and `transform` calls on the training data. Second, the transformed training data was passed on to the next object in the pipeline, `PCA`. Similar to the previous step, `PCA` also executed `fit` and `transform` on the scaled input data and passed it to the final element of the pipeline, the estimator.

Finally, the `LogisticRegression` estimator was fit to the training data after it underwent transformations via `StandardScaler` and `PCA`. Again, we should note that there is no limit to the number of intermediate steps in a pipeline; however, the last pipeline element has to be an estimator.

Similar to calling `fit` on a pipeline, pipelines also implement a `predict` method. If we feed a dataset to the `predict` call of a `Pipeline` object instance, the data will pass through the intermediate steps via `transform` calls. In the final step, the estimator object will then return a prediction on the transformed data.

The pipelines of the scikit-learn library are immensely useful wrapper tools, which we will use frequently throughout the rest of this book. To make sure that you've got a good grasp of how the `Pipeline` object works, please take a close look at the following illustration, which summarizes our discussion from the previous paragraphs:



Using k-fold cross-validation to assess model performance

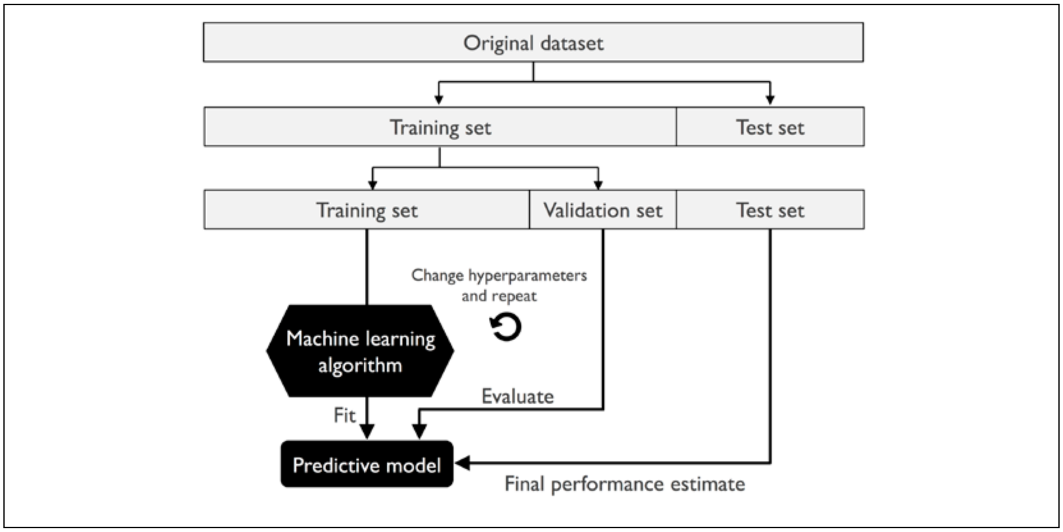
One of the key steps in building a machine learning model is to estimate its performance on data that the model hasn't seen before. Let's assume that we fit our model on a training dataset and use the same data to estimate how well it performs on new data. We remember from the *Tackling overfitting via regularization* section in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, that a model can suffer from underfitting (high bias) if the model is too simple, or it can overfit the training data (high variance) if the model is too complex for the underlying training data.

To find an acceptable bias-variance tradeoff, we need to evaluate our model carefully. In this section, you will learn about the common cross-validation techniques **holdout cross-validation** and **k-fold cross-validation**, which can help us to obtain reliable estimates of the model's generalization performance, that is, how well the model performs on unseen data.

The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, we split our initial dataset into separate training and test datasets—the former is used for model training, and the latter is used to estimate its generalization performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called **model selection**, with the name referring to a given classification problem for which we want to select the *optimal* values of *tuning parameters* (also called **hyperparameters**). However, if we reuse the same test dataset over and over again during model selection, it will become part of our training data and thus the model will be more likely to overfit. Despite this issue, many people still use the test dataset for model selection, which is not a good machine learning practice.

A better way of using the holdout method for model selection is to separate the data into three parts: a training dataset, a validation dataset, and a test dataset. The training dataset is used to fit the different models, and the performance on the validation dataset is then used for the model selection. The advantage of having a test dataset that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data. The following figure illustrates the concept of holdout cross-validation, where we use a validation dataset to repeatedly evaluate the performance of the model after training using different hyperparameter values. Once we are satisfied with the tuning of hyperparameter values, we estimate the model's generalization performance on the test dataset:



A disadvantage of the holdout method is that the performance estimate may be very sensitive to how we partition the training dataset into the training and validation subsets; the estimate will vary for different examples of the data. In the next subsection, we will take a look at a more robust technique for performance estimation, k-fold cross-validation, where we repeat the holdout method *k* times on *k* subsets of the training data.

K-fold cross-validation

In k-fold cross-validation, we randomly split the training dataset into *k* folds without replacement, where *k* – 1 folds are used for the model training, and one fold is used for performance evaluation. This procedure is repeated *k* times so that we obtain *k* models and performance estimates.

Sampling with and without replacement

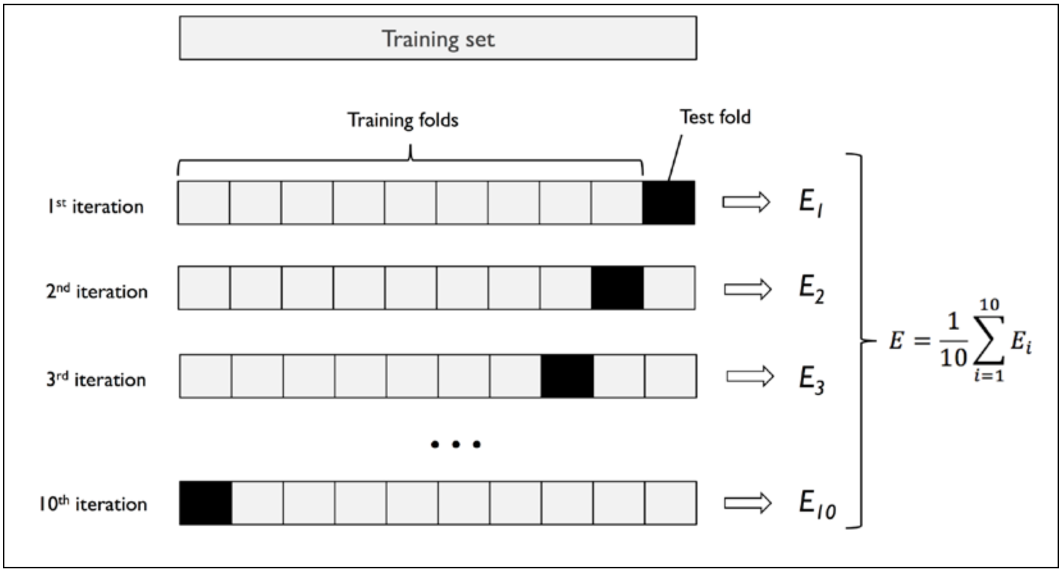
We looked at an example to illustrate sampling with and without replacement in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*. If you haven't read that chapter, or want a refresher, refer to the information box titled *Sampling with and without replacement* in the *Combining multiple decision trees via random forests* section.

We then calculate the average performance of the models based on the different, independent test folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance, which is estimated from evaluating the model performance on the test folds.

Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training dataset and obtain a final performance estimate using the independent test dataset. The rationale behind fitting a model to the whole training dataset after k-fold cross-validation is that providing more training examples to a learning algorithm usually results in a more accurate and robust model.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each example will be used for training and validation (as part of a test fold) exactly once, which yields a lower-variance estimate of the model performance than the holdout method. The following figure summarizes the concept behind k-fold cross-validation with *k* = 10. The training dataset is divided into 10 folds, and during the 10 iterations, nine folds are used for training, and one fold will be used as the test dataset for the model evaluation.

Also, the estimated performances, E_i (for example, classification accuracy or error), for each fold are then used to calculate the estimated average performance, *E*, of the model:



A good standard value for *k* in k-fold cross-validation is 10, as empirical evidence shows. For instance, experiments by Ron Kohavi on various real-world datasets suggest that 10-fold cross-validation offers the best tradeoff between bias and variance (*A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection*, *Kohavi, Ron*, *International Joint Conference on Artificial Intelligence (IJCAI)*, 14 (12): 1137-43, *1995*).

However, if we are working with relatively small training sets, it can be useful to increase the number of folds. If we increase the value of *k*, more training data will be used in each iteration, which results in a lower pessimistic bias toward estimating the generalization performance by averaging the individual model estimates. However, large values of *k* will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance, since the training folds will be more similar to each other. On the other hand, if we are working with large datasets, we can choose a smaller value for *k*, for example, *k* = 5, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

Leave-one-out cross-validation

A special case of k-fold cross-validation is the **leave-one-out cross-validation** (**LOOCV**) method. In LOOCV, we set the number of folds equal to the number of training examples ($k = n$) so that only one training example is used for testing during each iteration, which is a recommended approach for working with very small datasets.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, which has also been been shown in the same study by Ron Kohavi referenced previously in this section. In stratified cross-validation, the class label proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset, which we will illustrate by using the `StratifiedKFold` iterator in scikit-learn:

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold

>>> kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Fold: %2d, Class dist.: %s, Acc: %.3f' % (k+1,
...           np.bincount(y_train[train]), score))
Fold:  1, Class dist.: [256 153], Acc: 0.935
Fold:  2, Class dist.: [256 153], Acc: 0.935
Fold:  3, Class dist.: [256 153], Acc: 0.957
Fold:  4, Class dist.: [256 153], Acc: 0.957
Fold:  5, Class dist.: [256 153], Acc: 0.935
Fold:  6, Class dist.: [257 153], Acc: 0.956
Fold:  7, Class dist.: [257 153], Acc: 0.978
Fold:  8, Class dist.: [257 153], Acc: 0.933
Fold:  9, Class dist.: [257 153], Acc: 0.956
Fold: 10, Class dist.: [257 153], Acc: 0.956
>>> print('\nCV accuracy: %.3f +/- %.3f' %
...      (np.mean(scores), np.std(scores)))
CV accuracy: 0.950 +/- 0.014
```

First, we initialized the `StratifiedKFold` iterator from the `sklearn.model_selection` module with the `y_train` class labels in the training dataset, and we specified the number of folds via the `n_splits` parameter. When we used the `kfold` iterator to loop through the `k` folds, we used the returned indices in `train` to fit the logistic regression pipeline that we set up at the beginning of this chapter. Using the `pipe_lr` pipeline, we ensured that the examples were scaled properly (for instance, standardized) in each iteration. We then used the `test` indices to calculate the accuracy score of the model, which we collected in the `scores` list to calculate the average accuracy and the standard deviation of the estimate.

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows us to evaluate our model using stratified k-fold cross-validation less verbosely:

```
>>> from sklearn.model_selection import cross_val_score

>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print('CV accuracy scores: %s' % scores)
CV accuracy scores: [ 0.93478261  0.93478261  0.95652174
                      0.95652174  0.93478261  0.95555556
                      0.97777778  0.93333333  0.95555556
                      0.95555556]
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
...       np.std(scores)))
CV accuracy: 0.950 +/- 0.014
```

An extremely useful feature of the `cross_val_score` approach is that we can distribute the evaluation of the different folds across multiple central processing units (CPUs) on our machine. If we set the `n_jobs` parameter to `1`, only one CPU will be used to evaluate the performances, just like in our `StratifiedKFold` example previously. However, by setting `n_jobs=2`, we could distribute the 10 rounds of cross-validation to two CPUs (if available on our machine), and by setting `n_jobs=-1`, we can use all available CPUs on our machine to do the computation in parallel.

Estimating generalization performance

Please note that a detailed discussion of how the variance of the generalization performance is estimated in cross-validation is beyond the scope of this book, but you can refer to a comprehensive article about model evaluation and cross-validation (*Model evaluation, model selection, and algorithm selection in machine learning*. *Raschka S*. arXiv preprint arXiv:1811.12808, 2018) that discusses these topics in more depth. The article is freely available from `https://arxiv.org/abs/1811.12808`.

In addition, you can find a detailed discussion in this excellent article by M. Markatou and others (*Analysis of Variance of Cross-validation Estimators of the Generalization Error*, *M. Markatou*, *H. Tian*, *S. Biswas*, and *G. M. Hripcsak*, *Journal of Machine Learning Research*, 6: 1127-1168, *2005*).

You can also read about alternative cross-validation techniques, such as the .632 Bootstrap cross-validation method (*Improvements on Cross-validation: The .632+ Bootstrap Method*, *B. Efron* and *R. Tibshirani*, *Journal of the American Statistical Association*, 92(438): 548-560, *1997*).

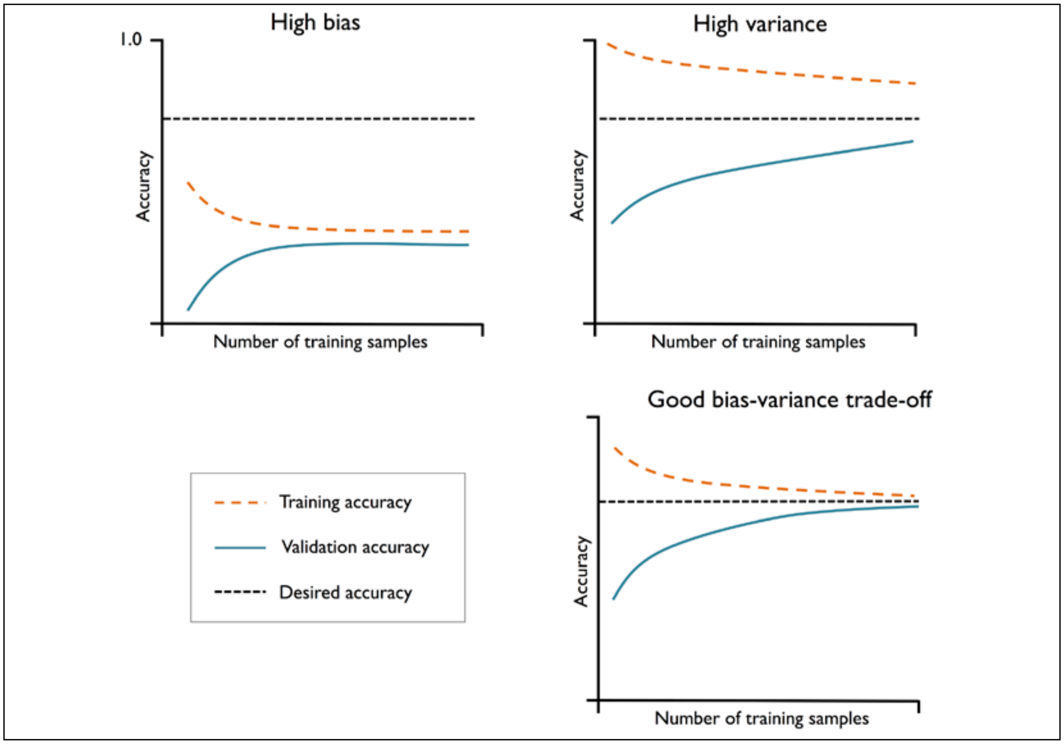
Debugging algorithms with learning and validation curves

In this section, we will take a look at two very simple yet powerful diagnostic tools that can help us to improve the performance of a learning algorithm: **learning curves** and **validation curves**. In the next subsections, we will discuss how we can use learning curves to diagnose whether a learning algorithm has a problem with overfitting (high variance) or underfitting (high bias). Furthermore, we will take a look at validation curves that can help us to address the common issues of a learning algorithm.

Diagnosing bias and variance problems with learning curves

If a model is too complex for a given training dataset—there are too many degrees of freedom or parameters in this model—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training examples to reduce the degree of overfitting.

However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training dataset size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help to address this problem. But before we discuss how to plot learning curves in scikit-learn, let's discuss those two common model issues by walking through the following illustration:



The graph in the upper-left shows a model with high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of parameters of the model, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in **support vector machine** (**SVM**) or logistic regression classifiers.

The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data, reduce the complexity of the model, or increase the regularization parameter, for example.

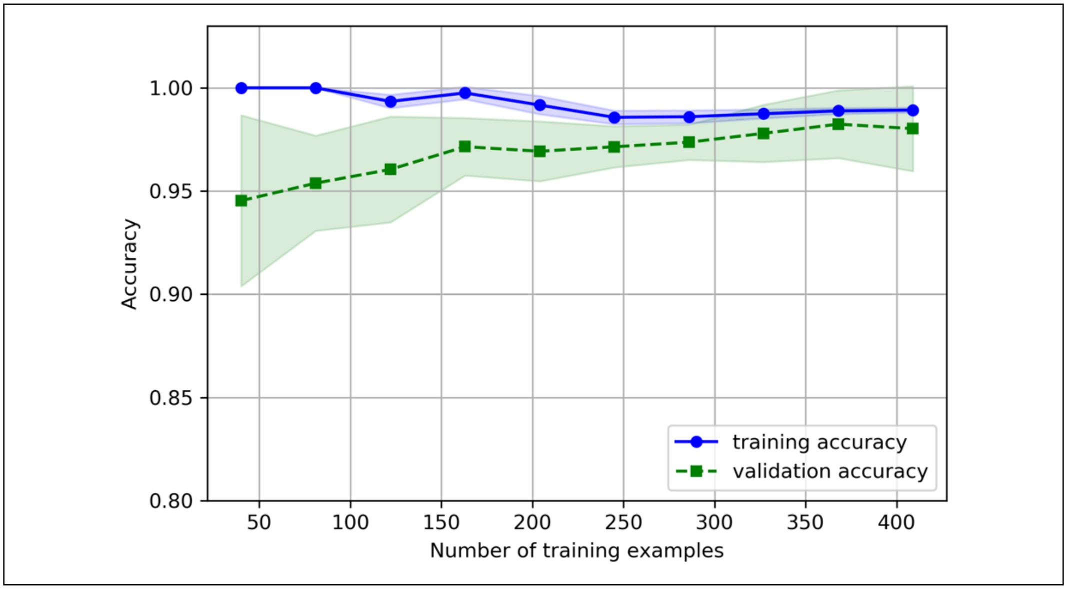
For unregularized models, it can also help to decrease the number of features via feature selection (*Chapter 4, Building Good Training Datasets – Data Preprocessing*) or feature extraction (*Chapter 5, Compressing Data via Dimensionality Reduction*) to decrease the degree of overfitting. While collecting more training data usually tends to decrease the chance of overfitting, it may not always help, for example, if the training data is extremely noisy or the model is already very close to optimal.

In the next subsection, we will see how to address those model issues using validation curves, but let's first see how we can use the learning curve function from scikit-learn to evaluate the model:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.model_selection import learning_curve

>>> pipe_lr = make_pipeline(StandardScaler(),
...                         LogisticRegression(penalty='l2',
...                                            random_state=1,
...                                            solver='lbfgs',
...                                            max_iter=10000))
>>> train_sizes, train_scores, test_scores =\
...                 learning_curve(estimator=pipe_lr,
...                                X=X_train,
...                                y=y_train,
...                                train_sizes=np.linspace(
...                                            0.1, 1.0, 10),
...                                cv=10,
...                                n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)

>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5, label='Training accuracy')

>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
...                  alpha=0.15, color='blue')

>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='Validation accuracy')
```

```
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training examples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.03])
>>> plt.show()
```

Note that we passed `max_iter=10000` as an additional argument when instantiating the `LogisticRegression` object (which uses 1,000 iterations as a default) to avoid convergence issues for the smaller dataset sizes or extreme regularization parameter values (covered in the next section). After we have successfully executed the preceding code, we will obtain the following learning curve plot:



Via the `train_sizes` parameter in the `learning_curve` function, we can control the absolute or relative number of training examples that are used to generate the learning curves. Here, we set `train_sizes=np.linspace(0.1, 1.0, 10)` to use 10 evenly spaced, relative intervals for the training dataset sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy of a classifier, and we set *k*=10 via the `cv` parameter for 10-fold stratified cross-validation.

Then, we simply calculated the average accuracies from the returned cross-validated training and test scores for the different sizes of the training dataset, which we plotted using Matplotlib's `plot` function. Furthermore, we added the standard deviation of the average accuracy to the plot using the `fill_between` function to indicate the variance of the estimate.

As we can see in the preceding learning curve plot, our model performs quite well on both the training and validation datasets if it has seen more than 250 examples during training. We can also see that the training accuracy increases for training datasets with fewer than 250 examples, and the gap between validation and training accuracy widens—an indicator of an increasing degree of overfitting.

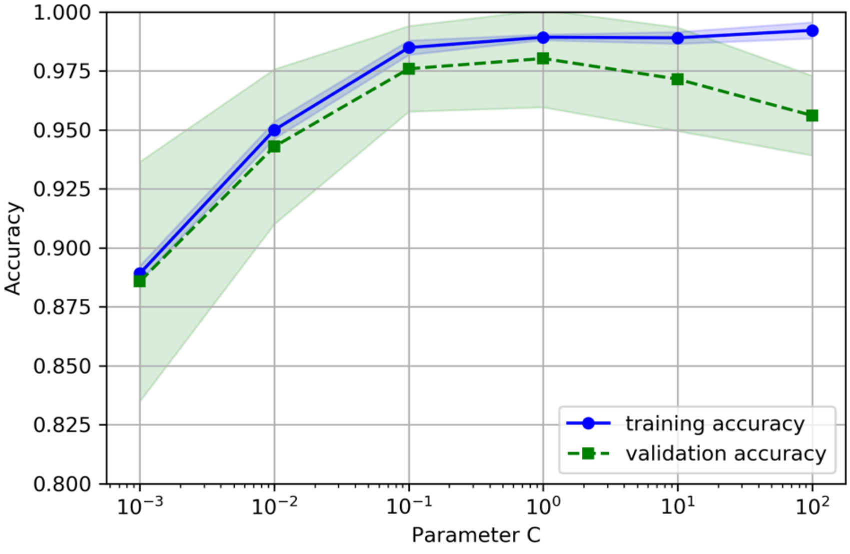
Addressing over- and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter, `C`, in logistic regression. Let's go ahead and see how we create validation curves via scikit-learn:

```
>>> from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                             estimator=pipe_lr,
...                             X=X_train,
...                             y=y_train,
...                             param_name='logisticregression__C',
...                             param_range=param_range,
...                             cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5, label='Training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='Validation accuracy')
```

```
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Using the preceding code, we obtained the validation curve plot for the parameter `C`:



Similar to the `learning_curve` function, the `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the classifier. Inside the `validation_curve` function, we specified the parameter that we wanted to evaluate. In this case, it is `C`, the inverse regularization parameter of the `LogisticRegression` classifier, which we wrote as `'logisticregression__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that we set via the `param_range` parameter. Similar to the learning curve example in the previous section, we plotted the average training and cross-validation accuracies and the corresponding standard deviations.

Although the differences in the accuracy for varying values of C are subtle, we can see that the model slightly underfits the data when we increase the regularization strength (small values of C). However, for large values of C, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be between 0.01 and 0.1 of the C value.

Fine-tuning machine learning models via grid search

In machine learning, we have two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters (or hyperparameters) of a model, for example, the regularization parameter in logistic regression or the depth parameter of a decision tree.

In the previous section, we used validation curves to improve the performance of a model by tuning one of its hyperparameters. In this section, we will take a look at a popular hyperparameter optimization technique called **grid search**, which can further help to improve the performance of a model by finding the *optimal* combination of hyperparameter values.

Tuning hyperparameters via grid search

The grid search approach is quite simple: it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination to obtain the optimal combination of values from this set:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import SVC

>>> pipe_svc = make_pipeline(StandardScaler(),
...                          SVC(random_state=1))
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'svc__C': param_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': param_range,
...                'svc__gamma': param_range,
...                'svc__kernel': ['rbf']}]

>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                      scoring='accuracy',
...                      cv=10,
...                      refit=True,
...                      n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.9846153846153847
>>> print(gs.best_params_)
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

Using the preceding code, we initialized a `GridSearchCV` object from the `sklearn.model_selection` module to train and tune an SVM pipeline. We set the `param_grid` parameter of `GridSearchCV` to a list of dictionaries to specify the parameters that we'd want to tune. For the linear SVM, we only evaluated the inverse regularization parameter, `C`; for the RBF kernel SVM, we tuned both the `svc__C` and `svc__gamma` parameters. Note that the `svc__gamma` parameter is specific to kernel SVMs.

After we used the training data to perform the grid search, we obtained the score of the best-performing model via the `best_score_` attribute and looked at its parameters, which can be accessed via the `best_params_` attribute. In this particular case, the RBF kernel SVM model with `svc__C = 100.0` yielded the best k-fold cross-validation accuracy: 98.5 percent.

Finally, we use the independent test dataset to estimate the performance of the best-selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print('Test accuracy: %.3f' % clf.score(X_test, y_test))
Test accuracy: 0.974
```

Please note that fitting a model with the best settings (`gs.best_estimator_`) on the training set manually via `clf.fit(X_train, y_train)` after completing the grid search is not necessary. The `GridSearchCV` class has a `refit` parameter, which will refit the `gs.best_estimator_` to the whole training set automatically if we set `refit=True` (default).

Randomized hyperparameter search

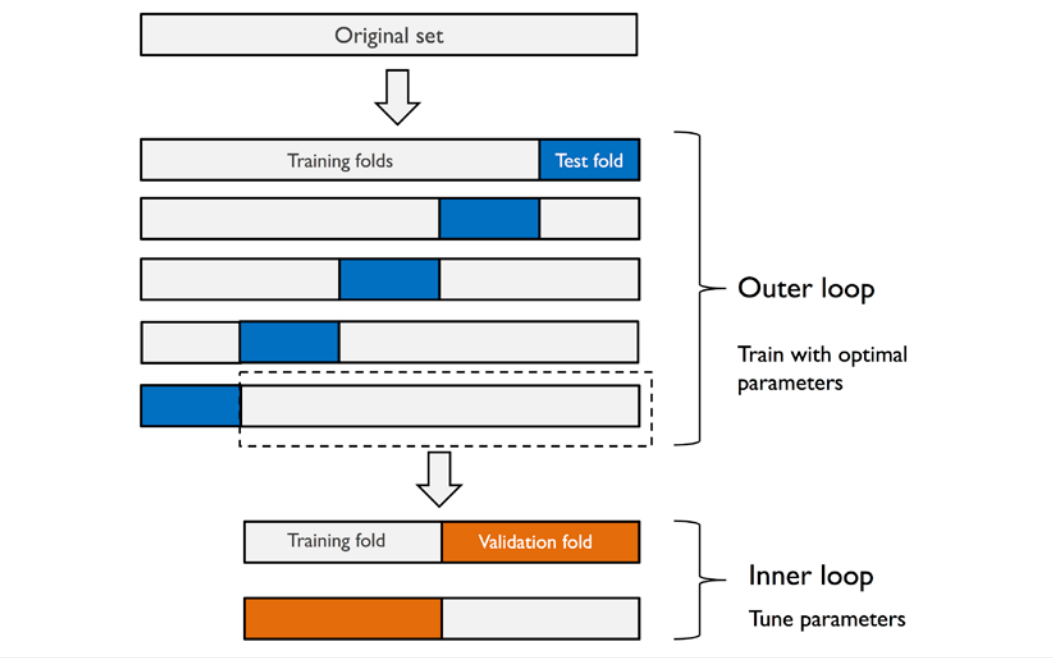
Although grid search is a powerful approach for finding the optimal set of parameters, the evaluation of all possible parameter combinations is also computationally very expensive. An alternative approach for sampling different parameter combinations using scikit-learn is *randomized search*. Randomized search usually performs about as well as grid search but is much more cost- and time-effective. In particular, if we only sample 60 parameter combinations via randomized search, we already have a 95 percent probability of obtaining solutions within 5 percent of the optimal performance (*Random search for hyper-parameter optimization*. *Bergstra J, Bengio Y*. *Journal of Machine Learning Research*. pp. 281-305, 2012).

Using the `RandomizedSearchCV` class in scikit-learn, we can draw random parameter combinations from sampling distributions with a specified budget. More details and examples of its usage can be found at `http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization`.

Algorithm selection with nested cross-validation

Using k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameter values, as we saw in the previous subsection. If we want to select among different machine learning algorithms, though, another recommended approach is nested cross-validation. In a nice study on the bias in error estimation, Sudhir Varma and Richard Simon concluded that the true error of the estimate is almost unbiased relative to the test dataset when nested cross-validation is used (*Bias in Error Estimation When Using Cross-Validation for Model Selection*, *BMC Bioinformatics*, *S. Varma* and *R. Simon*, 7(1): 91, *2006*).

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. The following figure explains the concept of nested cross-validation with only five outer and two inner folds, which can be useful for large datasets where computational performance is important; this particular type of nested cross-validation is also known as **5x2 cross-validation**:



In scikit-learn, we can perform nested cross-validation as follows:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2)

>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
...                                       np.std(scores)))
CV accuracy: 0.974 +/- 0.015
```

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and use it on unseen data.

For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier

>>> gs = GridSearchCV(estimator=DecisionTreeClassifier(
...                       random_state=0),
...                   param_grid=[{'max_depth': [1, 2, 3,
...                                             4, 5, 6,
...                                             7, None]}],
...                   scoring='accuracy',
...                   cv=2)

>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>>> print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores),
...                                       np.std(scores)))
CV accuracy: 0.934 +/- 0.016
```

As we can see, the nested cross-validation performance of the SVM model (97.4 percent) is notably better than the performance of the decision tree (93.4 percent), and thus, we'd expect that it might be the better choice to classify new data that comes from the same population as this particular dataset.

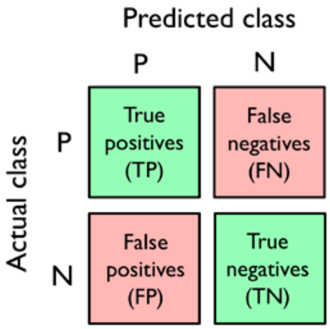
Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated different machine learning models using the prediction accuracy, which is a useful metric with which to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as precision, recall, and the **F1 score**.

Reading a confusion matrix

Before we get into the details of different scoring metrics, let's take a look at a **confusion matrix**, a matrix that lays out the performance of a learning algorithm.

A confusion matrix is simply a square matrix that reports the counts of the **true positive** (**TP**), **true negative** (**TN**), **false positive** (**FP**), and **false negative** (**FN**) predictions of a classifier, as shown in the following figure:



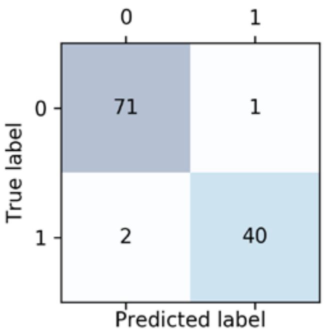
Although these metrics can be easily computed manually by comparing the true and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that we can use, as follows:

```
>>> from sklearn.metrics import confusion_matrix

>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
[ 2 40]]
```

The array that was returned after executing the code provides us with information about the different types of error the classifier made on the test dataset. We can map this information onto the confusion matrix illustration in the previous figure using Matplotlib's `matshow` function:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('Predicted label')
>>> plt.ylabel('True label')
>>> plt.show()
```

Now, the following confusion matrix plot, with the added labels, should make the results a little bit easier to interpret:



Assuming that class `1` (malignant) is the positive class in this example, our model correctly classified 71 of the examples that belong to class `0` (TN) and 40 examples that belong to class `1` (TP), respectively. However, our model also incorrectly misclassified two examples from class `1` as class `0` (FN), and it predicted that one example is malignant although it is a benign tumor (FP). In the next subsection, we will learn how we can use this information to calculate various error metrics.

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many examples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

The **true positive rate** (**TPR**) and **false positive rate** (**FPR**) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In tumor diagnosis, for example, we are more concerned about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors incorrectly classified as malignant (FP) to not unnecessarily concern patients. In contrast to the FPR, the TPR provides useful information about the fraction of positive (or relevant) examples that were correctly identified out of the total pool of positives (P).

The performance metrics **precision** (**PRE**) and **recall** (**REC**) are related to those TP and TN rates, and in fact, REC is synonymous with TPR:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Revisiting the malignant tumor detection example, optimizing for recall helps with minimizing the chance of not detecting a malignant tumor. However, this comes at the cost of predicting malignant tumors in patients although the patients are healthy (a high number of FP). If we optimize for precision, on the other hand, we emphasize correctness if we predict that a patient has a malignant tumor. However, this comes at the cost of missing malignant tumors more frequently (a high number of FN).

To balance the up- and down-sides of optimizing PRE and REC, often a combination of PRE and REC is used, the so-called F1 score:

$$F1 = 2\,\frac{PRE \times REC}{PRE + REC}$$

Further reading on precision and recall

If you are interested in a more thorough discussion of the different performance metrics, such as precision and recall, read David M. W. Powers' technical report *Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation*, which is freely available at `http://www.flinders.edu.au/science_engineering/fms/School-CSEM/publications/tech_reps-research_artfcts/TRRA_2007.pdf`.

Those scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module as shown in the following snippet:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score

>>> print('Precision: %.3f' % precision_score(
...           y_true=y_test, y_pred=y_pred))
Precision: 0.976
>>> print('Recall: %.3f' % recall_score(
...           y_true=y_test, y_pred=y_pred))
Recall: 0.952
>>> print('F1: %.3f' % f1_score(
...           y_true=y_test, y_pred=y_pred))
F1: 0.964
```

Furthermore, we can use a different scoring metric than accuracy in the `GridSearchCV` via the scoring parameter. A complete list of the different values that are accepted by the scoring parameter can be found at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Remember that the positive class in scikit-learn is the class that is labeled as class `1`. If we want to specify a different *positive label*, we can construct our own scorer via the `make_scorer` function, which we can then directly provide as an argument to the `scoring` parameter in `GridSearchCV` (in this example, using the `f1_score` as a metric):

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> c_gamma_range = [0.01, 0.1, 1.0, 10.0]
>>> param_grid = [{'svc__C': c_gamma_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': c_gamma_range,
...                'svc__gamma': c_gamma_range,
...                'svc__kernel': ['rbf']}]
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.986202145696
>>> print(gs.best_params_)
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

Plotting a receiver operating characteristic

Receiver operating characteristic (**ROC**) graphs are useful tools to select models for classification based on their performance with respect to the FPR and TPR, which are computed by shifting the decision threshold of the classifier. The diagonal of a ROC graph can be interpreted as *random guessing*, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a TPR of 1 and an FPR of 0. Based on the ROC curve, we can then compute the so-called **ROC area under the curve** (**ROC AUC**) to characterize the performance of a classification model.

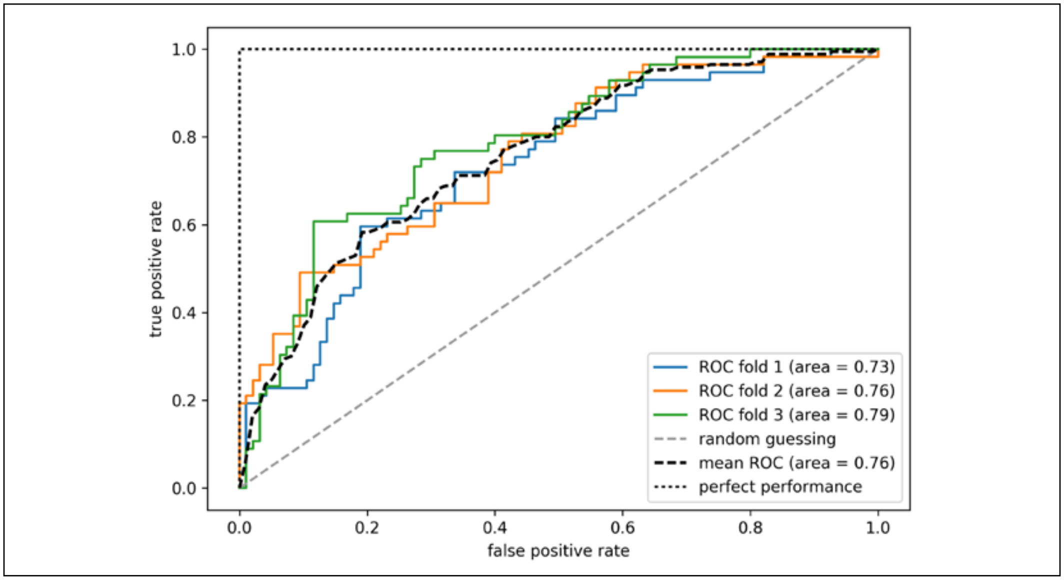
Similar to ROC curves, we can compute **precision-recall curves** for different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html.

Executing the following code example, we will plot a ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are only using two features this time. This is to make the classification task more challenging for the classifier, by witholding useful information contained in the other features, so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the StratifiedKFold validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp

>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression(penalty='l2',
...                                            random_state=1,
...                                            solver='lbfgs',
...                                            C=100.0))
```

```
>>> X_train2 = X_train[:, [4, 14]]

>>> cv = list(StratifiedKFold(n_splits=3,
...                           random_state=1).split(X_train,
...                                                 y_train))
>>> fig = plt.figure(figsize=(7, 5))

>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []

>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(
...         X_train2[train],
...         y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...              tpr,
...              label='ROC fold %d (area = %0.2f)'
...              % (i+1, roc_auc))
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='Random guessing')

>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          linestyle=':',
...          color='black',
...          label='Perfect performance')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('False positive rate')
>>> plt.ylabel('True positive rate')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from SciPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.76) falls between a perfect score (1.0) and random guessing (0.5):



Note that if we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule, which can be used similarly to the other scoring functions (for example, `precision_score`) that were introduced in the previous sections.

Reporting the performance of a classifier as the ROC AUC can yield further insights into a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cut-off point on an ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other: *The use of the area under the ROC curve in the evaluation of machine learning algorithms, A. P. Bradley, Pattern Recognition*, 30(7): 1145-1159, *1997*.

Scoring metrics for multiclass classification

The scoring metrics that we've discussed so far are specific to binary classification systems. However, scikit-learn also implements macro and micro averaging methods to extend those scoring metrics to multiclass problems via **one-vs.-all** (**OvA**) classification. The micro-average is calculated from the individual TPs, TNs, FPs, and FNs of the system. For example, the micro-average of the precision score in a *k*-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the `sklearn.metrics` module, for example, the `precision_score` or `make_scorer` functions:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

Dealing with class imbalance

We've mentioned class imbalances several times throughout this chapter, and yet we haven't actually discussed how to deal with such scenarios appropriately if they occur. Class imbalance is a quite common problem when working with real-world data—examples from one class or multiple classes are over-represented in a dataset. We can think of several domains where this may occur, such as spam filtering, fraud detection, or screening for diseases.

Imagine that the Breast Cancer Wisconsin dataset that we've been working with in this chapter consisted of 90 percent healthy patients. In this case, we could achieve 90 percent accuracy on the test dataset by just predicting the majority class (benign tumor) for all examples, without the help of a supervised machine learning algorithm. Thus, training a model on such a dataset that achieves approximately 90 percent test accuracy would mean our model hasn't learned anything useful from the features provided in this dataset.

In this section, we will briefly go over some of the techniques that could help with imbalanced datasets. But before we discuss different methods to approach this problem, let's create an imbalanced dataset from our dataset, which originally consisted of 357 benign tumors (class `0`) and 212 malignant tumors (class `1`):

```
>>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

In this code snippet, we took all 357 benign tumor examples and stacked them with the first 40 malignant examples to create a stark class imbalance. If we were to compute the accuracy of a model that always predicts the majority class (benign, class `0`), we would achieve a prediction accuracy of approximately 90 percent:

```
>>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.92443324937027
```

Thus, when we fit classifiers on such datasets, it would make sense to focus on other metrics than accuracy when comparing different models, such as precision, recall, the ROC curve—whatever we care most about in our application. For instance, our priority might be to identify the majority of patients with malignant cancer to recommend an additional screening, so recall should be our metric of choice. In spam filtering, where we don't want to label emails as spam if the system is not very certain, precision might be a more appropriate metric.

Aside from evaluating machine learning models, class imbalance influences a learning algorithm during model fitting itself. Since machine learning algorithms typically optimize a reward or cost function that is computed as a sum over the training examples that it sees during fitting, the decision rule is likely going to be biased toward the majority class.

In other words, the algorithm implicitly learns a model that optimizes the predictions based on the most abundant class in the dataset, in order to minimize the cost or maximize the reward during training.

One way to deal with imbalanced class proportions during model fitting is to assign a larger penalty to wrong predictions on the minority class. Via scikit-learn, adjusting such a penalty is as convenient as setting the `class_weight` parameter to `class_weight='balanced'`, which is implemented for most classifiers.

Other popular strategies for dealing with class imbalance include upsampling the minority class, downsampling the majority class, and the generation of synthetic training examples. Unfortunately, there's no universally best solution or technique that works best across different problem domains. Thus, in practice, it is recommended to try out different strategies on a given problem, evaluate the results, and choose the technique that seems most appropriate.

The scikit-learn library implements a simple `resample` function that can help with the upsampling of the minority class by drawing new samples from the dataset with replacement. The following code will take the minority class from our imbalanced Breast Cancer Wisconsin dataset (here, class `1`) and repeatedly draw new samples from it until it contains the same number of examples as class label `0`:

```
>>> from sklearn.utils import resample

>>> print('Number of class 1 examples before:',
...       X_imb[y_imb == 1].shape[0])
Number of class 1 examples before: 40

>>> X_upsampled, y_upsampled = resample(
...         X_imb[y_imb == 1],
...         y_imb[y_imb == 1],
...         replace=True,
...         n_samples=X_imb[y_imb == 0].shape[0],
...         random_state=123)
>>> print('Number of class 1 examples after:',
...       X_upsampled.shape[0])
Number of class 1 examples after: 357
```

After resampling, we can then stack the original class `0` samples with the upsampled class `1` subset to obtain a balanced dataset as follows:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Consequently, a majority vote prediction rule would only achieve 50 percent accuracy:

```
>>> y_pred = np.zeros(y_bal.shape[0])
>>> np.mean(y_pred == y_bal) * 100
50
```

Similarly, we could downsample the majority class by removing training examples from the dataset. To perform downsampling using the `resample` function, we could simply swap the class `1` label with class `0` in the previous code example and vice versa.

> **Generating new training data to address class-imbalance**
>
> Another technique for dealing with class imbalance is the generation of synthetic training examples, which is beyond the scope of this book. Probably the most widely used algorithm for synthetic training data generation is **Synthetic Minority Over-sampling Technique** (**SMOTE**), and you can learn more about this technique in the original research article by Nitesh Chawla and others: *SMOTE: Synthetic Minority Over-sampling Technique*, *Journal of Artificial Intelligence Research*, 16: 321-357, *2002*. It is also highly recommended to check out `imbalanced-learn`, a Python library that is entirely focused on imbalanced datasets, including an implementation of SMOTE. You can learn more about `imbalanced-learn` at `https://github.com/scikit-learn-contrib/imbalanced-learn`.

Summary

At the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that help us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose common problems of learning algorithms, such as overfitting and underfitting.

Using grid search, we further fine-tuned our model. We then used confusion matrices and various performance metrics to evaluate and optimize a model's performance for specific problem tasks. Finally, we concluded this chapter by discussing different methods for dealing with imbalanced data, which is a common problem in many real-world applications. Now, you should be well-equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will look at ensemble methods: methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.

7

Combining Different Models for Ensemble Learning

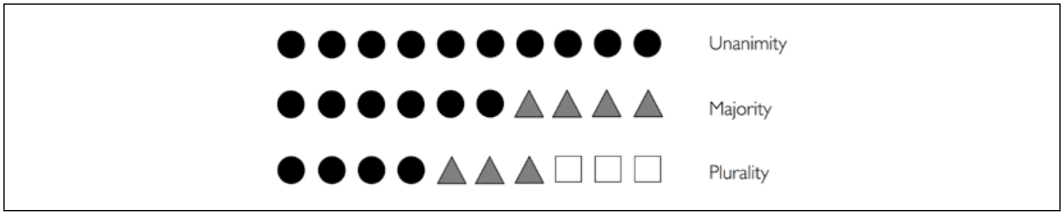
In the previous chapter, we focused on the best practices for tuning and evaluating different models for classification. In this chapter, we will build upon those techniques and explore different methods for constructing a set of classifiers that can often have a better predictive performance than any of its individual members. We will learn how to do the following:

- Make predictions based on majority voting
- Use bagging to reduce overfitting by drawing random combinations of the training dataset with repetition
- Apply boosting to build powerful models from weak learners that learn from their mistakes

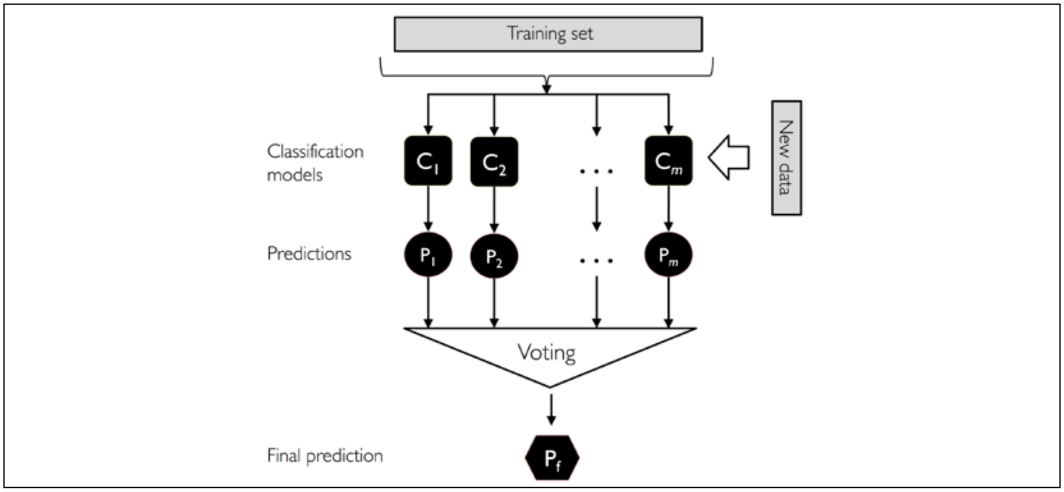
Learning with ensembles

The goal of **ensemble methods** is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine those predictions by the 10 experts to come up with a prediction that was more accurate and robust than the predictions by each individual expert. As you will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. This section will introduce a basic explanation of how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term "majority vote" refers to binary class settings only. However, it is easy to generalize the majority voting principle to multiclass settings, which is called **plurality voting**. Here, we select the class label that received the most votes (the mode). The following diagram illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers, where each unique symbol (triangle, square, and circle) represents a unique class label:



Using the training dataset, we start by training m different classifiers $(C_1, \ldots, C_m)$. Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm, fitting different subsets of the training dataset. One prominent example of this approach is the random forest algorithm, which combines different decision tree classifiers. The following figure illustrates the concept of a general ensemble approach using majority voting:



To predict a class label via simple majority or plurality voting, we can combine the predicted class labels of each individual classifier, C_j, and select the class label, $\hat{y}$, that received the most votes:

$$\hat{y} = \text{mode}\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

(In statistics, the mode is the most frequent event or result in a set. For example, mode{1, 2, 1, 1, 2, 4, 5, 4} = 1.)

For example, in a binary classification task where class1 = –1 and class2 = +1, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = \text{ sign}\left[\sum_{j}^{m} C_j(\boldsymbol{x})\right] = \begin{cases} 1 & \text{if } \sum_j C_j(\boldsymbol{x}) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we will make the assumption that all *n*-base classifiers for a binary classification task have an equal error rate, ε. Furthermore, we will assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \ \geq k) = \sum_{k}^{n} \left\langle {n \atop k} \right\rangle \varepsilon^k (1-\varepsilon)^{n-k} = \ \varepsilon_{\text{ensemble}}$$

Here, $\left\langle {n \atop k} \right\rangle$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now, let's take a look at a more concrete example of 11 base classifiers (*n* = 11), where each classifier has an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \ \geq k) = \sum_{k=6}^{11} \left\langle {11 \atop k} \right\rangle 0.25^k (1-0.25)^{11-k} = \ 0.034$$

The binomial coefficient

The binomial coefficient refers to the number of ways we can choose subsets of k-unordered elements from a set of size n; thus, it is often called "n choose k." Since the order does not matter here, the binomial coefficient is also sometimes referred to as *combination* or *combinatorial number*, and in its unabbreviated form, it is written as follows:

$$\frac{n!}{(n-k)!\,k!}$$

Here, the symbol (!) stands for factorial—for example, $3! = 3 \cdot 2 \cdot 1 = 6$.

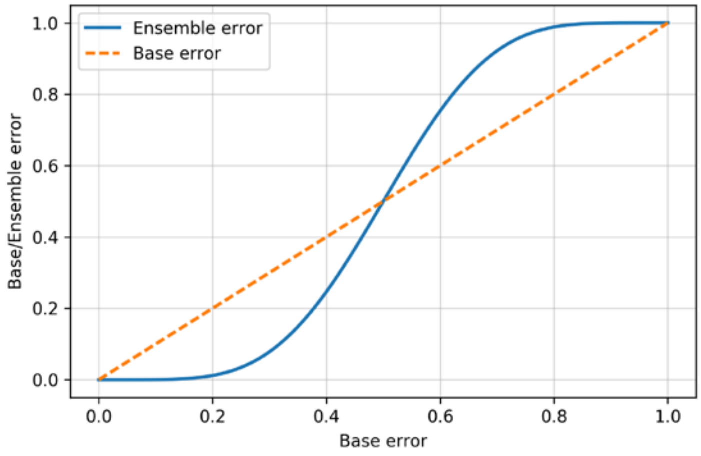
As you can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers, n, is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.special import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

After we have implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...              linewidth=2)
>>> plt.plot(error_range, error_range,
...              linestyle='--', label='Base error',
...              linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid(alpha=0.5)
>>> plt.show()
```

As you can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier, as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$). Note that the *y* axis depicts the base error (dotted line) as well as the ensemble error (continuous line):



Combining classifiers via majority vote

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python.

Plurality voting

Although the majority voting algorithm that we will discuss in this section also generalizes to multiclass settings via plurality voting, the term "majority voting" will be used for simplicity, as is often the case in the literature.

Implementing a simple majority vote classifier

The algorithm that we are going to implement in this section will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A\big(C_j(\boldsymbol{x}) = i\big)$$

Here, w_j is a weight associated with a base classifier, C_j; $\hat{y}$ is the predicted class label of the ensemble; A is the set of unique class labels; χ_A (Greek chi) is the characteristic function or indicator function, which returns 1 if the predicted class of the *j*th classifier matches i ($C_j(\boldsymbol{x}) = i$). For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = \text{mode}\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers, $C_j(j \in \{0, 1\})$, and we want to predict the class label, $C_j(\boldsymbol{x}) \in \{0,1\}$, of a given example, $\boldsymbol{x}$. Two out of three base classifiers predict the class label 0, and one, C_3, predicts that the example belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote predicts that the example belongs to class 0:

$$C_1(\boldsymbol{x}) \rightarrow 0, \quad C_2(\boldsymbol{x}) \rightarrow 0, \quad C_3(\boldsymbol{x}) \rightarrow 1$$

$$\hat{y} = \text{mode}\{0,0,1\} = 0$$

Now, let's assign a weight of 0.6 to C_3, and let's weight C_1 and C_2 by a coefficient of 0.2:

$$\begin{aligned} \hat{y} \quad &= \quad \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A\big(C_j(\boldsymbol{x}) = i\big) \\ &= \quad \arg\max_{i}[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1 \end{aligned}$$

More simply, since $3 \times 0.2 = 0.6$, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, which we can write as follows:

$$\hat{y} = \text{mode}\{0,0,1,1,1\} = 1$$

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...           weights=[0.2, 0.2, 0.6]))
1
```

As you will remember from the discussion on logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the *j*th classifier for class label *i*.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0, 1\}$ and an ensemble of three classifiers, $C_j (j \in \{1, 2, 3\})$. Let's assume that the classifiers C_j return the following class membership probabilities for a particular example, $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1], \quad C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2], \quad C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

Using the same weights as previously (0.2, 0.2, and 0.6), we can then calculate the individual class probabilities as follows:

$$\begin{aligned} p(i_0|\boldsymbol{x}) &= 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58 \\ p(i_1|\boldsymbol{x}) &= 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42 \\ \hat{y} &= \arg\max_{i} [p(i_0|\,\boldsymbol{x}), p(i_1|\,\boldsymbol{x})] = 0 \end{aligned}$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy, using `np.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([0.58, 0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement `MajorityVoteClassifier` in Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of 'int' or 'float' values are
      provided, the classifiers are weighted by
      importance; Uses uniform weights if 'weights=None'.

    """
    def __init__(self, classifiers,
                 vote='classlabel', weights=None):
```

```
        self.classifiers = classifiers
        self.named_classifiers = {key: value for
                                  key, value in
                                  _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Fit classifiers.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_examples, n_features]
            Matrix of training examples.

        y : array-like, shape = [n_examples]
            Vector of target class labels.

        Returns
        -------
        self : object

        """
        if self.vote not in ('probability', 'classlabel'):
            raise ValueError("vote must be 'probability'"
                             "or 'classlabel'; got (vote=%r)"
                             % self.vote)

        if self.weights and
        len(self.weights) != len(self.classifiers):
            raise ValueError("Number of classifiers and weights"
                             "must be equal; got %d weights,"
                             "%d classifiers"
                             % (len(self.weights),
                             len(self.classifiers)))
        # Use LabelEncoder to ensure class labels start
        # with 0, which is important for np.argmax
        # call in self.predict
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                               self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

I've added a lot of comments to the code to explain the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the BaseEstimator and ClassifierMixin parent classes to get some base functionality *for free*, including the get_params and set_params methods to set and return the classifier's parameters, as well as the score method to calculate the prediction accuracy.

Next, we will add the predict method to predict the class label via a majority vote based on the class labels if we initialize a new MajorityVoteClassifier object with vote='classlabel'. Alternatively, we will be able to initialize the ensemble classifier with vote='probability' to predict the class label based on the class membership probabilities. Furthermore, we will also add a predict_proba method to return the averaged probabilities, which is useful when computing the receiver operating characteristic area under the curve (ROC AUC):

```
    def predict(self, X):
        """ Predict class labels for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            Shape = [n_examples, n_features]
            Matrix of training examples.

        Returns
        ----------
        maj_vote : array-like, shape = [n_examples]
            Predicted class labels.

        """
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X), axis=1)
        else: # 'classlabel' vote

            # Collect results from clf.predict calls
            predictions = np.asarray([clf.predict(X)
                                      for clf in
                                      self.classifiers_]).T

            maj_vote = np.apply_along_axis(lambda x: np.argmax(
                                           np.bincount(x,
                                           weights=self.weights)),
                                           axis=1,
                                           arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_examples, n_features]
            Training vectors, where
            n_examples is the number of examples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like,
            shape = [n_examples, n_classes]
            Weighted average probability for
            each class per example.

        """
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas, axis=0,
                               weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
            return super(MajorityVoteClassifier,
                           self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in self.named_classifiers.items():
                for key, value in step.get_params(
                        deep=True).items():
                    out['%s__%s' % (name, key)] = value
            return out
```

Also, note that we defined our own modified version of the `get_params` method to use the `_name_estimators` function to access the parameters of individual classifiers in the ensemble; this may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter tuning in later sections.

VotingClassifier in scikit-learn

Although the `MajorityVoteClassifier` implementation is very useful for demonstration purposes, we implemented a more sophisticated version of this majority vote classifier in scikit-learn based on the implementation in the first edition of this book. The ensemble classifier is available as `sklearn.ensemble.VotingClassifier` in scikit-learn version 0.17 and newer.

Using the majority voting principle to make predictions

Now it is time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the Iris dataset from scikit-learn's `datasets` module. Furthermore, we will only select two features, *sepal width* and *petal length*, to make the classification task more challenging for illustration purposes. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower examples from the `Iris-versicolor` and `Iris-virginica` classes, with which we will compute the ROC AUC later. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Class membership probabilities from decision trees

Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then, the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that these are actually not derived from probability mass functions.

Next, we will split the Iris examples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.5,
...                      random_state=1,
...                      stratify=y)
```

Using the training dataset, we now will train three different classifiers:

- Logistic regression classifier
- Decision tree classifier
- k-nearest neighbors classifier

We will then evaluate the model performance of each classifier via 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           solver='lbfgs',
...                           random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic regression', 'Decision tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a pipeline. The reason behind it is that, as discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant, in contrast to decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now, let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...                 classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
ROC AUC: 0.98 (+/- 0.05) [Majority voting]
```

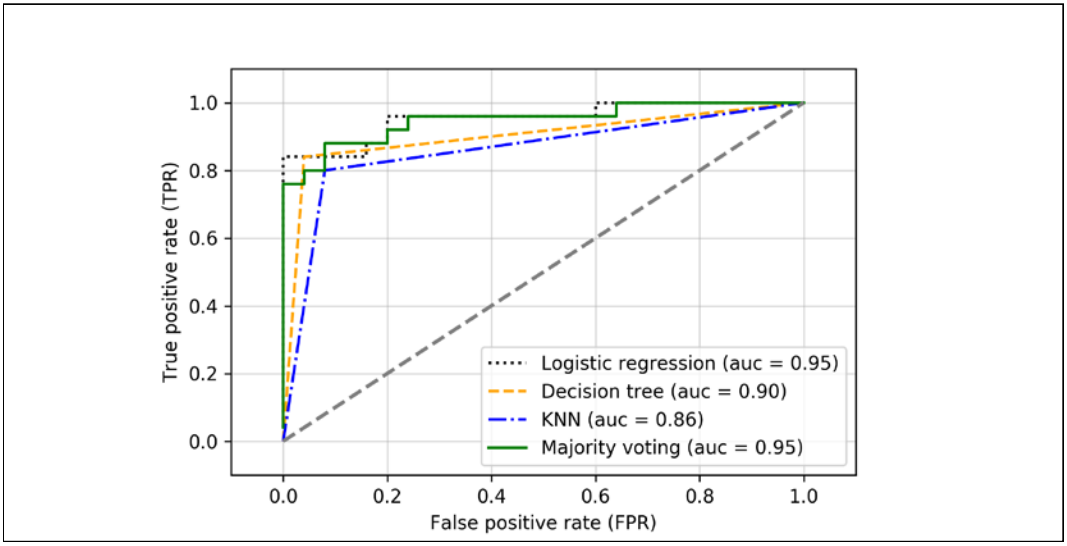
As you can see, the performance of `MajorityVotingClassifier` has improved over the individual classifiers in the 10-fold cross-validation evaluation.

Evaluating and tuning the ensemble classifier

In this section, we are going to compute the ROC curves from the test dataset to check that `MajorityVoteClassifier` generalizes well with unseen data. We must remember that the test dataset is not to be used for model selection; its purpose is merely to report an unbiased estimate of the generalization performance of a classifier system:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...                linestyle=ls,
...                label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
>>> plt.xlabel('False positive rate (FPR)')
>>> plt.ylabel('True positive rate (TPR)')
>>> plt.show()
```

As you can see in the resulting ROC, the ensemble classifier also performs well on the test dataset (ROC AUC = 0.95). However, you can see that the logistic regression classifier performs similarly well on the same dataset, which is probably due to the high variance (in this case, sensitivity of how we split the dataset) given the small size of the dataset:

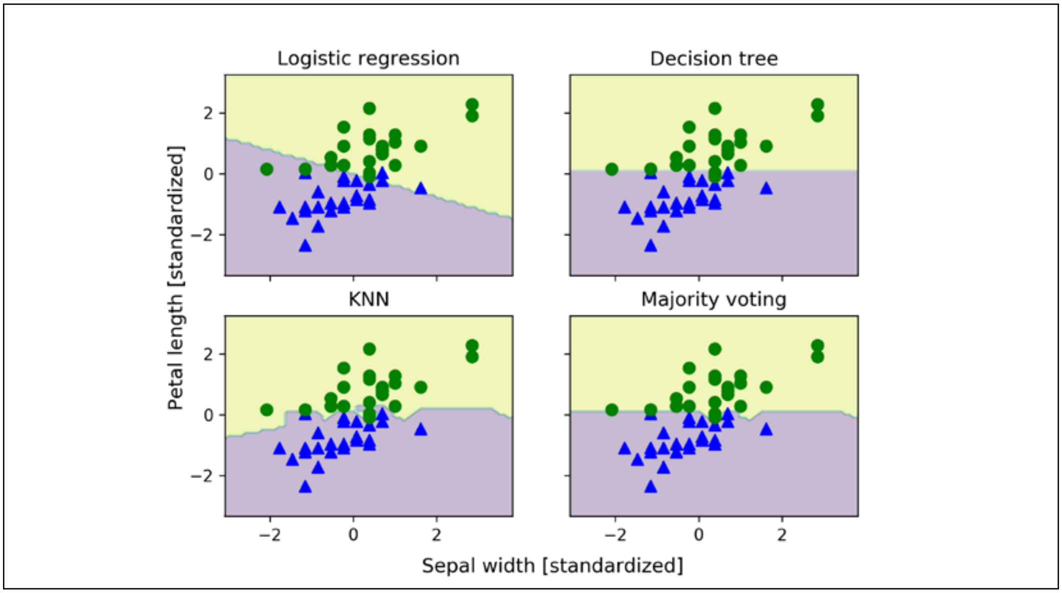


Since we only selected two features for the classification examples, it would be interesting to see what the decision region of the ensemble classifier actually looks like.

Although it is not necessary to standardize the training features prior to model fitting, because our logistic regression and k-nearest neighbors pipelines will automatically take care of it, we will standardize the training dataset so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='green',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -5.,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-12.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

Interestingly, but also as expected, the decision regions of the ensemble classifier seem to be a hybrid of the decision regions from the individual classifiers. At first glance, the majority vote decision boundary looks a lot like the decision of the decision tree stump, which is orthogonal to the *y* axis for *sepal width* ≥ 1. However, you can also notice the non-linearity from the k-nearest neighbor classifier mixed in:



Before we tune the individual classifier's parameters for ensemble classification, let's call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearchCV` object:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier':
 DecisionTreeClassifier(class_weight=None, criterion='entropy',
                        max_depth=1, max_features=None,
                        max_leaf_nodes=None, min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-1':
```

```
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                      with_std=True)),
                ('clf', LogisticRegression(C=0.001,
                                           class_weight=None,
                                           dual=False,
                                           fit_intercept=True,
                                           intercept_scaling=1,
                                           max_iter=100,
                                           multi_class='ovr',
                                           penalty='l2',
                                           random_state=0,
                                           solver='liblinear',
                                           tol=0.0001,
                                           verbose=0))]),
'pipeline-1__clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1,
                   max_iter=100, multi_class='ovr',
                   penalty='l2', random_state=0,
                   solver='liblinear', tol=0.0001, verbose=0),
'pipeline-1__clf__C': 0.001,
'pipeline-1__clf__class_weight': None,
'pipeline-1__clf__dual': False,
[...]
'pipeline-1__sc__with_std': True,
'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                      with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                           leaf_size=30,
                                           metric='minkowski',
                                           metric_params=None,
                                           n_neighbors=1,
                                           p=2,
                                           weights='uniform'))]),
'pipeline-2__clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                     metric='minkowski', metric_params=None,
                     n_neighbors=1, p=2, weights='uniform'),
'pipeline-2__clf__algorithm': 'auto',
[...]
'pipeline-2__sc__with_std': True}
```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Let's now tune the inverse regularization parameter, `C`, of the logistic regression classifier and the decision tree depth via a grid search for demonstration purposes:

```
>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     iid=False,
...                     scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

After the grid search has completed, we can print the different hyperparameter value combinations and the average ROC AUC scores computed via 10-fold cross-validation as follows:

```
>>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     print("%0.3f +/- %0.2f %r"
...           % (grid.cv_results_['mean_test_score'][r],
...              grid.cv_results_['std_test_score'][r] / 2.0,
...              grid.cv_results_['params'][r]))
0.944 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 0.001}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 100.0}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 0.001}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 100.0}

>>> print('Best parameters: %s' % grid.best_params_)
Best parameters: {'decisiontreeclassifier__max_depth': 1,
                  'pipeline-1__clf__C': 0.001}

>>> print('Accuracy: %.2f' % grid.best_score_)
Accuracy: 0.98
```

As you can see, we get the best cross-validation results when we choose a lower regularization strength (`C=0.001`), whereas the tree depth does not seem to affect the performance at all, suggesting that a decision stump is sufficient to separate the data. To remind ourselves that it is a bad practice to use the test dataset more than once for model evaluation, we are not going to estimate the generalization performance of the tuned hyperparameters in this section. We will move on swiftly to an alternative approach for ensemble learning: **bagging**.

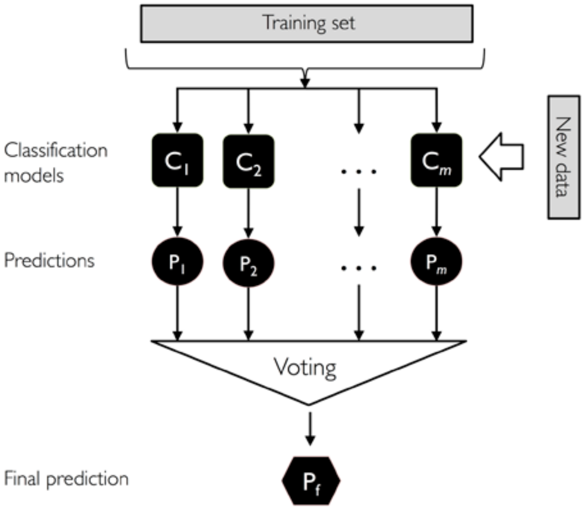
Building ensembles using stacking

The majority vote approach we implemented in this section is not to be confused with **stacking**. The stacking algorithm can be understood as a two-level ensemble, where the first level consists of individual classifiers that feed their predictions to the second level, where another classifier (typically logistic regression) is fit to the level-one classifier predictions to make the final predictions. The stacking algorithm has been described in more detail by David H. Wolpert in *Stacked generalization*, *Neural Networks*, 5(2):241–259, *1992*. Unfortunately, an implementation of this algorithm has not been implemented in scikit-learn at the time of writing; however, this feature is underway. In the meantime, you can find scikit-learn-compatible implementations of stacking at `http://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/` and `http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/`.

Bagging – building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the `MajorityVoteClassifier` that we implemented in the previous section. However, instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training dataset, which is why bagging is also known as *bootstrap aggregating*.

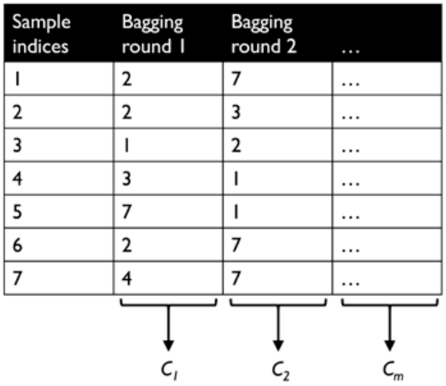
The concept of bagging is summarized in the following diagram:



In the following subsections, we will work through a simple example of bagging by hand and use scikit-learn for classifying wine examples.

Bagging in a nutshell

To provide a more concrete example of how the bootstrap aggregating of a bagging classifier works, let's consider the example shown in the following figure. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier, C_j, which is most typically an unpruned decision tree:



Sample indices	Bagging round 1	Bagging round 2	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...

As you can see from the previous illustration, each classifier receives a random subset of examples from the training dataset. We denote these random samples obtained via bagging as *Bagging round 1*, *Bagging round 2*, and so on. Each subset contains a certain portion of duplicates and some of the original examples don't appear in a resampled dataset at all due to sampling with replacement. Once the individual classifiers are fit to the bootstrap samples, the predictions are combined using majority voting.

Note that bagging is also related to the random forest classifier that we introduced in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*. In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

Model ensembles using bagging

Bagging was first proposed by Leo Breiman in a technical report in 1994; he also showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting. I highly recommend that you read about his research in *Bagging predictors*, *L. Breiman*, *Machine Learning*, 24(2):123–140, *1996*, which is freely available online, to learn more details about bagging.

Applying bagging to classify examples in the Wine dataset

To see bagging in action, let's create a more complex classification problem using the Wine dataset that was introduced in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we will select two features – `Alcohol` and `OD280/OD315 of diluted wines`:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...               'machine-learning-databases/wine/wine.data',
...               header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> # drop 1 class
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol',
...              'OD280/OD315 of diluted wines']].values
```

Next, we will encode the class labels into binary format and split the dataset into 80 percent training and 20 percent test datasets, respectively:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...            train_test_split(X, y,
...                             test_size=0.2,
...                             random_state=1,
...                             stratify=y)
```

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable. For instance, to load the Wine dataset from a local directory, take the following lines:

```
df = pd.read_csv(
         'https://archive.ics.uci.edu/ml/'
         'machine-learning-databases'
         '/wine/wine.data', header=None)
```

and replace them with these:

```
df = pd.read_csv(
         'your/local/path/to/wine.data',
         header=None)
```

A `BaggingClassifier` algorithm is already implemented in scikit-learn, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fit on different bootstrap samples of the training dataset:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

Next, we will calculate the accuracy score of the prediction on the training and test datasets to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 1.000/0.833
```

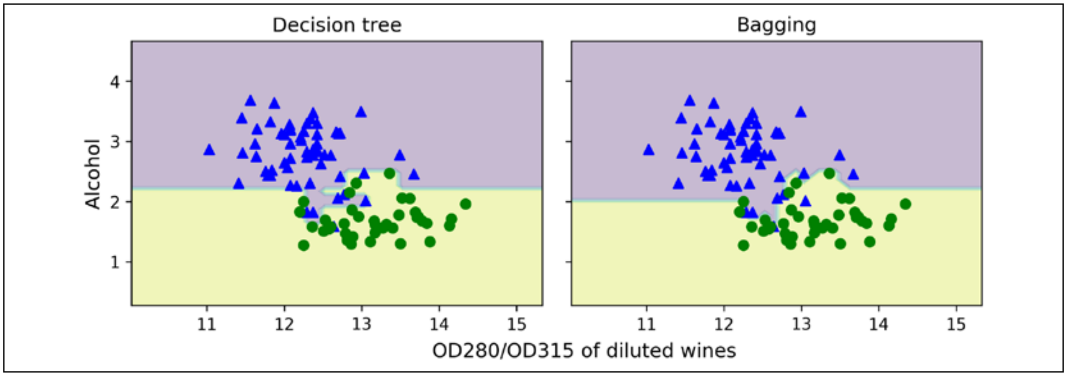
Based on the accuracy values that we printed here, the unpruned decision tree predicts all the class labels of the training examples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print('Bagging train/test accuracies %.3f/%.3f'
...       % (bag_train, bag_test))
Bagging train/test accuracies 1.000/0.917
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training dataset (both 100 percent), we can see that the bagging classifier has a slightly better generalization performance, as estimated on the test dataset. Next, let's compare the decision regions between the decision tree and the bagging classifier:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Alcohol', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='OD280/OD315 of diluted wines',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

As we can see in the resulting plot, the piece-wise linear decision boundary of the three-node deep decision tree looks smoother in the bagging ensemble:



We only looked at a very simple bagging example in this section. In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths. Finally, we must note that the bagging algorithm can be an effective approach to reducing the variance of a model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trend in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

Leveraging weak learners via adaptive boosting

In this last section about ensemble methods, we will discuss **boosting**, with a special focus on its most common implementation: **Adaptive Boosting** (**AdaBoost**).

AdaBoost recognition

The original idea behind AdaBoost was formulated by Robert E. Schapire in 1990. *The Strength of Weak Learnability, R. E. Schapire*, *Machine Learning*, 5(2): 197-227, *1990*. After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the *Proceedings of the Thirteenth International Conference* (ICML 1996), AdaBoost became one of the most widely used ensemble methods in the years that followed (*Experiments with a New Boosting Algorithm by Y. Freund*, *R. E. Schapire*, and others, *ICML*, volume 96, 148-156, *1996*). In 2003, Freund and Schapire received the Goedel Prize for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the field of computer science.

In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing—a typical example of a weak learner is a decision tree stump. The key concept behind boosting is to focus on training examples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance of the ensemble.

The following subsections will introduce the algorithmic procedure behind the general concept of boosting and AdaBoost. Lastly, we will use scikit-learn for a practical classification example.

How boosting works

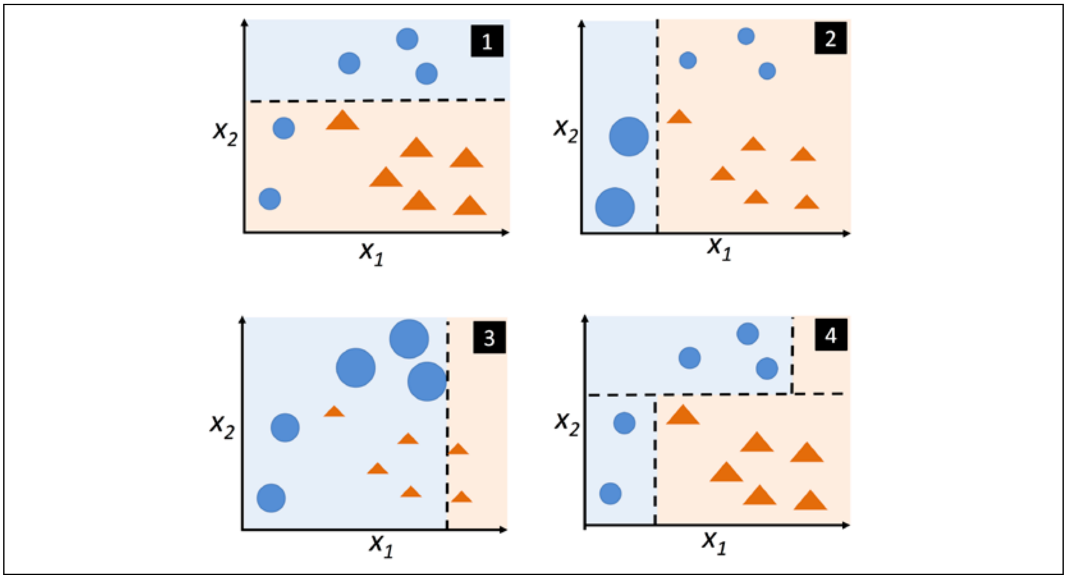
In contrast to bagging, the initial formulation of the boosting algorithm uses random subsets of training examples drawn from the training dataset without replacement; the original boosting procedure can be summarized in the following four key steps:

1. Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, *D*, to train a weak learner, C_1.
2. Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, C_2.
3. Find the training examples, d_3, in the training dataset, *D*, which C_1 and C_2 disagree upon, to train a third weak learner, C_3.
4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

As discussed by Leo Breiman (*Bias, variance, and arcing classifiers*, *L. Breiman*, *1996*), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (*An improvement of AdaBoost to avoid overfitting*, *G. Raetsch*, *T. Onoda*, and *K. R. Mueller*. *Proceedings of the International Conference on Neural Information Processing*, *CiteSeer*, *1998*).

In contrast to the original boosting procedure described here, AdaBoost uses the complete training dataset to train the weak learners, where the training examples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.

Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at the following figure to get a better grasp of the basic concept behind AdaBoost:



To walk through the AdaBoost illustration step by step, we will start with subfigure **1**, which represents a training dataset for binary classification where all training examples are assigned equal weights. Based on this training dataset, we train a decision stump (shown as a dashed line) that tries to classify the examples of the two classes (triangles and circles), as well as possibly minimizing the cost function (or the impurity score in the special case of decision tree ensembles).

For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified examples (circles). Furthermore, we lower the weight of the correctly classified examples. The next decision stump will now be more focused on the training examples that have the largest weights—the training examples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different examples from the circle class, which are then assigned a larger weight, as shown in subfigure **3**.

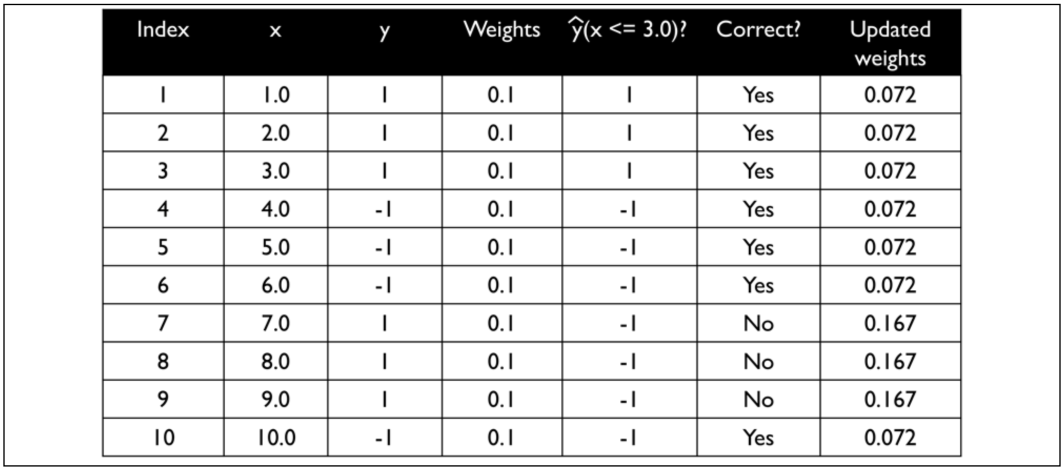
Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that we have a better understanding of the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol (×) and the dot-product between two vectors by a dot symbol (·):

```
1. Set the weight vector, $\boldsymbol{w}$, to uniform weights, where $\sum_{\boldsymbol{i}} w_i = 1$.
2. For $j$ in $m$ boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{\boldsymbol{y}} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
    d. Compute coefficient: $\alpha_j = 0.5 \log \frac{1 - \boldsymbol{\varepsilon}}{\boldsymbol{\varepsilon}}$
    e. Update weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
    f. Normalize weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left( \sum_{j=1}^{m} \left( \boldsymbol{\alpha}_j \times \text{predict}(C_j, \boldsymbol{X}) \right) > 0 \right)$.
```

Note that the expression $(\hat{\boldsymbol{y}} \neq \boldsymbol{y})$ in step 2c refers to a binary vector consisting of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training dataset consisting of 10 training examples, as illustrated in the following table:



Index	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The first column of the table depicts the indices of training examples 1 to 10. In the second column, you can see the feature values of the individual samples, assuming this is a one-dimensional dataset. The third column shows the true class label, y_i, for each training sample, x_i, where $y_i \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights uniformly (assigning the same constant value) and normalize them to sum to 1. In the case of the 10-sample training dataset, we therefore assign 0.1 to each weight, w_i, in the weight vector, $\boldsymbol{w}$. The predicted class labels, $\hat{\boldsymbol{y}}$, are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudo code.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We will start by computing the weighted error rate, ε, as described in step 2c:

$$\begin{aligned}\varepsilon \quad = \quad & 0.1 \times 0 + \ 0.1 \times 0 + \ 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 \\ & +0.1 \times 1 + 0.1 \times 1 + 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3\end{aligned}$$

Next, we will compute the coefficient, α_j—shown in step 2d—which will later be used in step 2e to update the weights, as well as for the weights in the majority vote prediction (step 3):

$$\alpha_j = 0.5 \log\left(\frac{1-\varepsilon}{\varepsilon}\right) \approx 0.424$$

After we have computed the coefficient, α_j, we can now update the weight vector using the following equation:

$$\boldsymbol{w} := \boldsymbol{w} \ \times \exp(-\alpha_j \ \times \ \hat{\boldsymbol{y}} \ \times \ \boldsymbol{y})$$

Here, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction, $\hat{y}_i$, is correct, $\hat{\boldsymbol{y}}_{\boldsymbol{i}} \times \boldsymbol{y}_{\boldsymbol{i}}$ will have a positive sign so that we decrease the *i*th weight, since α_j is a positive number as well:

$$0.1 \times \ \exp(-0.424 \ \times \ 1 \ \times 1) \approx 0.065$$

Similarly, we will increase the *i*th weight if $\hat{y}_i$ predicted the label incorrectly, like this:

$$0.1 \times \ \exp(-0.424 \ \times \ 1 \ \times (-1)) \approx 0.153$$

Alternatively, it's like this:

$$0.1 \ \times \ \exp(-0.424 \ \times \ (-1) \ \times (1)) \approx 0.153$$

After we have updated each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 2f):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.

Thus, each weight that corresponds to a correctly classified example will be reduced from the initial value of 0.1 to $0.065/0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of the incorrectly classified examples will increase from 0.1 to $0.153/0.914 \approx 0.167$.

Applying AdaBoost using scikit-learn

The previous subsection introduced AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier. Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Decision tree train/test accuracies %.3f/%.3f'
...       % (tree_train, tree_test))
Decision tree train/test accuracies 0.916/0.875
```

As you can see, the decision tree stump seems to underfit the training data in contrast to the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('AdaBoost train/test accuracies %.3f/%.3f'
...       % (ada_train, ada_test))
AdaBoost train/test accuracies 1.000/0.917
```

Here, you can see that the AdaBoost model predicts all class labels of the training dataset correctly and also shows a slightly improved test dataset performance compared to the decision tree stump. However, you can also see that we introduced additional variance by our attempt to reduce the model bias—a greater gap between training and test performance.

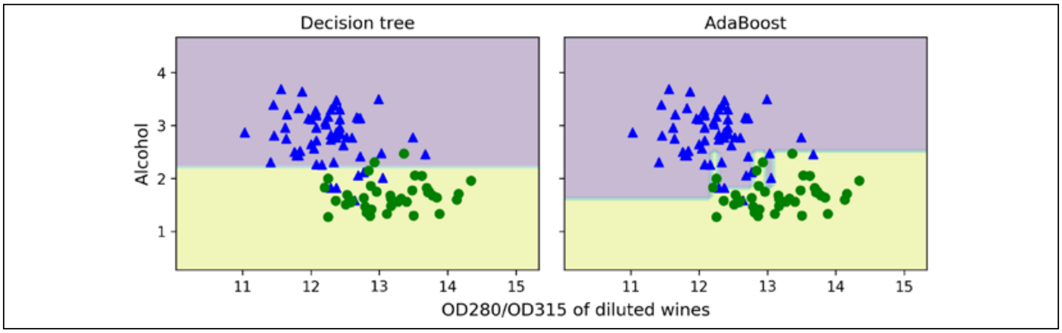
Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores as the bagging classifier that we trained in the previous section. However, we must note that it is considered bad practice to select a model based on the repeated usage of the test dataset. The estimate of the generalization performance may be over-optimistic, which we discussed in more detail in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Lastly, let's check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision Tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Alcohol', fontsize=12)
```

```
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='OD280/OD315 of diluted wines',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

By looking at the decision regions, you can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section:



As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computational costs for an often relatively modest improvement in predictive performance.

An often-cited example of this tradeoff is the famous $1 million *Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in *The BigChaos Solution to the Netflix Grand Prize* by *A. Toescher*, *M. Jahrer*, and *R. M. Bell*, *Netflix Prize documentation*, *2009*, which is available at `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf`. The winning team received the $1 million grand prize money; however, Netflix never implemented their model due to its complexity, which made it infeasible for a real-world application:

> *"We evaluated some of the new methods offline but the additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment."*
>
> *http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html*

Gradient boosting

Another popular variant of boosting is **gradient boosting**. AdaBoost and gradient boosting share the main overall concept: boosting weak learners (such as decision tree stumps) to strong learners. The two approaches, adaptive and gradient boosting, differ mainly with regard to how the weights are updated and how the (weak) classifiers are combined. If you are familiar with gradient-based optimization and interested in gradient boosting, I recommend reading Jerome Friedman's work (*Greedy function approximation: a gradient boosting machine*. *Jerome Friedman*. *Annals of Statistics 2001*, pp.1189-1232) and the more recent paper on XGBoost, which is essentially a computationally efficient implementation of the original gradient boost algorithm (*XGBoost: A scalable tree boosting system*. *Tianqi Chen* and *Carlos Guestrin*. *Proceeding of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. *ACM 2016*, pp. 785-794). Note that next to the `GradientBoostingClassifier` implementation in scikit-learn, scikit-learn now also includes a substantially faster version of gradient boosting in version 0.21, `HistGradientBoostingClassifier`, which is even faster than XGBoost. For more information about the `GradientBoostingClassifier` and `HistGradientBoostingClassifier` in scikit-learn, you can read the documentation at `https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting`. Also, a short, general explanation of gradient boosting can be found in the lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles__notes.pdf`

Summary

In this chapter, we looked at some of the most popular and widely used techniques for ensemble learning. Ensemble methods combine different classification models to cancel out their individual weaknesses, which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions.

At the beginning of this chapter, we implemented `MajorityVoteClassifier` in Python, which allows us to combine different algorithms for classification. We then looked at bagging, a useful technique for reducing the variance of a model by drawing random bootstrap samples from the training dataset and combining the individually trained classifiers via majority vote. Lastly, we learned about AdaBoost, which is an algorithm that is based on weak learners that subsequently learn from mistakes.

Throughout the previous chapters, we learned a lot about different learning algorithms, tuning, and evaluation techniques. In the next chapter, we will look at a particular application of machine learning, sentiment analysis, which has become an interesting topic in the internet and social media era.

8

Applying Machine Learning to Sentiment Analysis

In the modern internet and social media age, people's opinions, reviews, and recommendations have become a valuable resource for political science and businesses. Thanks to modern technologies, we are now able to collect and analyze such data most efficiently. In this chapter, we will delve into a subfield of **natural language processing** (**NLP**) called **sentiment analysis** and learn how to use machine learning algorithms to classify documents based on their polarity: the attitude of the writer. In particular, we are going to work with a dataset of 50,000 movie reviews from the **Internet Movie Database** (**IMDb**) and build a predictor that can distinguish between positive and negative reviews.

The topics that we will cover in the following sections include the following:

- Cleaning and preparing text data
- Building feature vectors from text documents
- Training a machine learning model to classify positive and negative movie reviews
- Working with large text datasets using out-of-core learning
- Inferring topics from document collections for categorization

Preparing the IMDb movie review data for text processing

As mentioned, sentiment analysis, sometimes also called **opinion mining**, is a popular subdiscipline of the broader field of NLP; it is concerned with analyzing the polarity of documents. A popular task in sentiment analysis is the classification of documents based on the expressed opinions or emotions of the authors with regard to a particular topic.

In this chapter, we will be working with a large dataset of movie reviews from the Internet Movie Database (IMDb) that has been collected by Andrew Maas and others (*Learning Word Vectors for Sentiment Analysis*, *A. L. Maas*, *R. E. Daly*, *P. T. Pham*, *D. Huang*, *A. Y. Ng*, and *C. Potts*, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, Association for Computational Linguistics, *June 2011*). The movie review dataset consists of 50,000 polar movie reviews that are labeled as either positive or negative; here, positive means that a movie was rated with more than six stars on IMDb, and negative means that a movie was rated with fewer than five stars on IMDb. In the following sections, we will download the dataset, preprocess it into a useable format for machine learning tools, and extract meaningful information from a subset of these movie reviews to build a machine learning model that can predict whether a certain reviewer liked or disliked a movie.

Obtaining the movie review dataset

A compressed archive of the movie review dataset (84.1 MB) can be downloaded from `http://ai.stanford.edu/~amaas/data/sentiment/` as a gzip-compressed tarball archive:

- If you are working with Linux or macOS, you can open a new terminal window, `cd` into the download directory and execute `tar -zxf aclImdb_v1.tar.gz` to decompress the dataset.
- If you are working with Windows, you can download a free archiver, such as 7-Zip (`http://www.7-zip.org`), to extract the files from the download archive.
- Alternatively, you can directly unpack the gzip-compressed tarball archive directly in Python as follows:

```
>>> import tarfile
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
...     tar.extractall()
```

Preprocessing the movie dataset into a more convenient format

Having successfully extracted the dataset, we will now assemble the individual text documents from the decompressed download archive into a single CSV file. In the following code section, we will be reading the movie reviews into a pandas `DataFrame` object, which can take up to 10 minutes on a standard desktop computer.

To visualize the progress and estimated time until completion, we will use the **Python Progress Indicator** (**PyPrind**, `https://pypi.python.org/pypi/PyPrind/`) package, which was developed several years ago for such purposes. PyPrind can be installed by executing the `pip install pyprind` command:

```
>>> import pyprind
>>> import pandas as pd
>>> import os

>>> # change the 'basepath' to the directory of the
>>> # unzipped movie dataset

>>> basepath = 'aclImdb'
>>>
>>> labels = {'pos': 1, 'neg': 0}
>>> pbar = pyprind.ProgBar(50000)
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for file in sorted(os.listdir(path)):
...             with open(os.path.join(path, file),
...                       'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...             df = df.append([[txt, labels[l]]],
...                            ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                          100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:02:05
```

In the preceding code, we first initialized a new progress bar object, `pbar`, with 50,000 iterations, which was the number of documents we were going to read in. Using the nested `for` loops, we iterated over the `train` and `test` subdirectories in the main `aclImdb` directory and read the individual text files from the `pos` and `neg` subdirectories that we eventually appended to the `df` pandas `DataFrame`, together with an integer class label (`1` = positive and `0` = negative).

Since the class labels in the assembled dataset are sorted, we will now shuffle `DataFrame` using the `permutation` function from the `np.random` submodule—this will be useful to split the dataset into training and test datasets in later sections, when we will stream the data from our local drive directly.

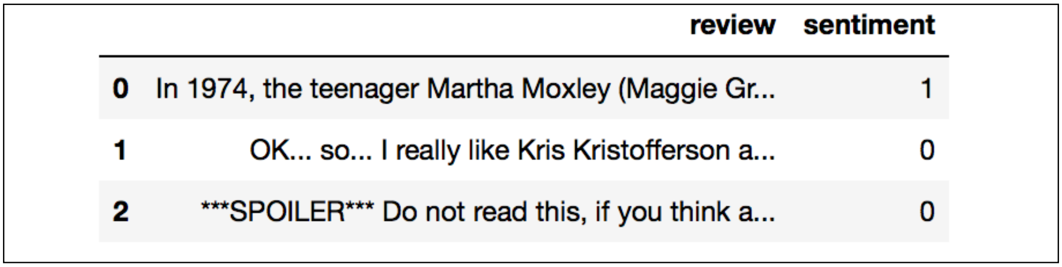
For our own convenience, we will also store the assembled and shuffled movie review dataset as a CSV file:

```
>>> import numpy as np

>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Since we are going to use this dataset later in this chapter, let's quickly confirm that we have successfully saved the data in the right format by reading in the CSV and printing an excerpt of the first three examples:

```
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>>> df.head(3)
```

If you are running the code examples in a Jupyter Notebook, you should now see the first three examples of the dataset, as shown in the following table:



	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

As a sanity check, before we proceed to the next section, let's make sure that the `DataFrame` contains all 50,000 rows:

```
>>> df.shape
(50000, 2)
```

Introducing the bag-of-words model

You may remember from *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, that we have to convert categorical data, such as text or words, into a numerical form before we can pass it on to a machine learning algorithm. In this section, we will introduce the **bag-of-words** model, which allows us to represent text as numerical feature vectors. The idea behind bag-of-words is quite simple and can be summarized as follows:

1. We create a vocabulary of unique tokens—for example, words—from the entire set of documents.

2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

Since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will mostly consist of zeros, which is why we call them **sparse**. Do not worry if this sounds too abstract; in the following subsections, we will walk through the process of creating a simple bag-of-words model step by step.

Transforming words into feature vectors

To construct a bag-of-words model based on the word counts in the respective documents, we can use the `CountVectorizer` class implemented in scikit-learn. As you will see in the following code section, `CountVectorizer` takes an array of text data, which can be documents or sentences, and constructs the bag-of-words model for us:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array(['The sun is shining',
...                  'The weather is sweet',
...                  'The sun is shining, the weather is sweet,'
...                  'and one and one is two'])
>>> bag = count.fit_transform(docs)
```

By calling the `fit_transform` method on `CountVectorizer`, we constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

- `'The sun is shining'`
- `'The weather is sweet'`
- `'The sun is shining, the weather is sweet, and one and one is two'`

Now, let's print the contents of the vocabulary to get a better understanding of the underlying concepts:

```
>>> print(count.vocabulary_)
{'and': 0,
'two': 7,
'shining': 3,
'one': 2,
'sun': 4,
'weather': 8,
'the': 6,
'sweet': 5,
'is': 1}
```

As you can see from executing the preceding command, the vocabulary is stored in a Python dictionary that maps the unique words to integer indices. Next, let's print the feature vectors that we just created:

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the `CountVectorizer` vocabulary. For example, the first feature at index position `0` resembles the count of the word `'and'`, which only occurs in the last document, and the word `'is'`, at index position `1` (the second feature in the document vectors), occurs in all three sentences. These values in the feature vectors are also called the **raw term frequencies**: *tf*(*t*, *d*)—the number of times a term, *t*, occurs in a document, *d*. It should be noted that, in the bag-of-words model, the word or term order in a sentence or document does not matter. The order in which the term frequencies appear in the feature vector is derived from the vocabulary indices, which are usually assigned alphabetically.

N-gram models

The sequence of items in the bag-of-words model that we just created is also called the **1-gram** or **unigram** model—each item or token in the vocabulary represents a single word. More generally, the contiguous sequences of items in NLP—words, letters, or symbols—are also called **n-grams**. The choice of the number, *n*, in the n-gram model depends on the particular application; for example, a study by Ioannis Kanaris and others revealed that n-grams of size 3 and 4 yield good performances in the anti-spam filtering of email messages (*Words versus character n-grams for anti-spam filtering*, *Ioannis Kanaris*, *Konstantinos Kanaris*, *Ioannis Houvardas*, and *Efstathios Stamatatos*, *International Journal on Artificial Intelligence Tools*, *World Scientific Publishing Company*, 16(06): 1047-1067, *2007*).

To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document "the sun is shining" would be constructed as follows:

- 1-gram: "the", "sun", "is", "shining"
- 2-gram: "the sun", "sun is", "is shining"

The `CountVectorizer` class in scikit-learn allows us to use different n-gram models via its `ngram_range` parameter. While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new `CountVectorizer` instance with `ngram_range=(2,2)`.

Assessing word relevancy via term frequency-inverse document frequency

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. These frequently occurring words typically don't contain useful or discriminatory information. In this subsection, you will learn about a useful technique called the **term frequency-inverse document frequency** (**tf-idf**), which can be used to downweight these frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$tf\text{-}idf(t,d) = tf(t,d) \times idf(t,d)$$

Here, *tf*(*t*, *d*) is the term frequency that we introduced in the previous section, and *idf*(*t*, *d*) is the inverse document frequency, which can be calculated as follows:

$$idf(t,d) = \log \frac{\mathrm{n_d}}{1 + df(d,t)}$$

Here, $\mathrm{n_d}$ is the total number of documents, and *df*(*d*, *t*) is the number of documents, *d*, that contain the term *t*. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in none of the training examples; the *log* is used to ensure that low document frequencies are not given too much weight.

The scikit-learn library implements yet another transformer, the `TfidfTransformer` class, which takes the raw term frequencies from the `CountVectorizer` class as input and transforms them into tf-idfs:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer(use_idf=True,
...                          norm='l2',
...                          smooth_idf=True)
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs))
...       .toarray())
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

As you saw in the previous subsection, the word `'is'` had the largest term frequency in the third document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, the word `'is'` is now associated with a relatively small tf-idf (`0.45`) in the third document, since it is also present in the first and second document and thus is unlikely to contain any useful discriminatory information.

However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we would have noticed that `TfidfTransformer` calculates the tf-idfs slightly differently compared to the standard textbook equations that we defined previously. The equation for the inverse document frequency implemented in scikit-learn is computed as follows:

$$idf(t,d) = \log\frac{1+\mathrm{n_d}}{1+df(d,t)}$$

Similarly, the tf-idf computed in scikit-learn deviates slightly from the default equation we defined earlier:

$$tf\text{-}idf(t,d) = tf(t,d) \times (idf(t,d)+1)$$

Note that the "+1" in the previous equations is due to setting `smooth_idf=True` in the previous code example, which is helpful for assigning zero-weight (that is, $idf(t, d) = \log(1) = 0$) to terms that occur in all documents.

While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the `TfidfTransformer` class normalizes the tf-idfs directly. By default (`norm='l2'`), scikit-learn's `TfidfTransformer` applies the L2-normalization, which returns a vector of length 1 by dividing an unnormalized feature vector, v, by its L2-norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{{v_1}^2 + {v_2}^2 + \cdots + {v_n}^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

To make sure that we understand how `TfidfTransformer` works, let's walk through an example and calculate the tf-idf of the word `'is'` in the third document. The word `'is'` has a term frequency of 3 (*tf* = 3) in the third document, and the document frequency of this term is 3 since the term `'is'` occurs in all three documents (*df* = 3). Thus, we can calculate the inverse document frequency as follows:

$$idf(\text{"is"}, d_3) = \log \frac{1+3}{1+3} = 0$$

Now, in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$tf\text{-}idf(\text{"is"}, d_3) = 3 \times (0 + 1) = 3$$

If we repeated this calculation for all terms in the third document, we'd obtain the following tf-idf vectors: `[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]`. However, notice that the values in this feature vector are different from the values that we obtained from `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\begin{aligned} tf\text{-}idf(d_3)_{norm} &= \frac{[3.39,\ 3.0,\ 3.39,\ 1.29,\ 1.29,\ 1.29,\ 2.0,\ 1.69,\ 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \\ tf\text{-}idf(\text{"is"}, d_3) &= 0.45 \end{aligned}$$

As you can see, the results now match the results returned by scikit-learn's `TfidfTransformer`, and since you now understand how tf-idfs are calculated, let's proceed to the next section and apply those concepts to the movie review dataset.

Cleaning text data

In the previous subsections, we learned about the bag-of-words model, term frequencies, and tf-idfs. However, the first important step—before we build our bag-of-words model—is to clean the text data by stripping it of all unwanted characters.

To illustrate why this is important, let's display the last 50 characters from the first document in the reshuffled movie review dataset:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

As you can see here, the text contains HTML markup as well as punctuation and other non-letter characters. While HTML markup does not contain many useful semantics, punctuation marks can represent useful, additional information in certain NLP contexts. However, for simplicity, we will now remove all punctuation marks except for emoticon characters, such as :), since those are certainly useful for sentiment analysis. To accomplish this task, we will use Python's **regular expression** (**regex**) library, `re`, as shown here:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text)
...     text = (re.sub('[\W]+', ' ', text.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     return text
```

Via the first regex, `<[^>]*>`, in the preceding code section, we tried to remove all of the HTML markup from the movie reviews. Although many programmers generally advise against the use of regex to parse HTML, this regex should be sufficient to *clean* this particular dataset. Since we are only interested in removing HTML markup and do not plan to use the HTML markup further, using regex to do the job should be acceptable. However, if you prefer using sophisticated tools for removing HTML markup from text, you can take a look at Python's HTML parser module, which is described at `https://docs.python.org/3/library/html.parser.html`. After we removed the HTML markup, we used a slightly more complex regex to find emoticons, which we temporarily stored as emoticons. Next, we removed all non-word characters from the text via the regex `[\W]+` and converted the text into lowercase characters.

Dealing with word capitalization

In the context of this analysis, we assume that the capitalization of a word—for example, whether it appears at the beginning of a sentence—does not contain semantically relevant information. However, note that there are exceptions; for instance, we remove the notation of proper names. But again, in the context of this analysis, it is a simplifying assumption that the letter case does not contain information that is relevant for sentiment analysis.

Eventually, we added the temporarily stored emoticons to the end of the processed document string. Additionally, we removed the *nose* character (- in :-)) from the emoticons for consistency.

> **Regular expressions**
>
> Although regular expressions offer an efficient and convenient approach to searching for characters in a string, they also come with a steep learning curve. Unfortunately, an in-depth discussion of regular expressions is beyond the scope of this book. However, you can find a great tutorial on the Google Developers portal at `https://developers.google.com/edu/python/regular-expressions` or you can check out the official documentation of Python's `re` module at `https://docs.python.org/3.7/library/re.html`.

Although the addition of the emoticon characters to the end of the cleaned document strings may not look like the most elegant approach, we must note that the order of the words doesn't matter in our bag-of-words model if our vocabulary consists of only one-word tokens. But before we talk more about the splitting of documents into individual terms, words, or tokens, let's confirm that our `preprocessor` function works correctly:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Lastly, since we will make use of the *cleaned* text data over and over again during the next sections, let's now apply our `preprocessor` function to all the movie reviews in our `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

Processing documents into tokens

After successfully preparing the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to *tokenize* documents is to split them into individual words by splitting the cleaned documents at their whitespace characters:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

In the context of tokenization, another useful technique is **word stemming**, which is the process of transforming a word into its root form. It allows us to map related words to the same stem. The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the **Porter stemmer** algorithm (*An algorithm for suffix stripping, Martin F. Porter, Program: Electronic Library and Information Systems*, 14(3): 130–137, *1980*). The **Natural Language Toolkit** (**NLTK**, `http://www.nltk.org`) for Python implements the Porter stemming algorithm, which we will use in the following code section. In order to install the NLTK, you can simply execute `conda install nltk` or `pip install nltk`.

NLTK online book

Although the NLTK is not the focus of this chapter, I highly recommend that you visit the NLTK website as well as read the official NLTK book, which is freely available at `http://www.nltk.org/book/`, if you are interested in more advanced applications in NLP.

The following code shows how to use the Porter stemming algorithm:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Using the `PorterStemmer` from the `nltk` package, we modified our `tokenizer` function to reduce words to their root form, which was illustrated by the simple preceding example where the word `'running'` was *stemmed* to its root form `'run'`.

Stemming algorithms

The Porter stemming algorithm is probably the oldest and simplest stemming algorithm. Other popular stemming algorithms include the newer **Snowball stemmer** (Porter2 or English stemmer) and the **Lancaster stemmer** (Paice/Husk stemmer). While both the Snowball and Lancaster stemmers are faster than the original Porter stemmer, the Lancaster stemmer is also notorious for being more aggressive than the Porter stemmer. These alternative stemming algorithms are also available through the NLTK package (`http://www.nltk.org/api/nltk.stem.html`).

> While stemming can create non-real words, such as `'thu'` (from `'thus'`), as shown in the previous example, a technique called **lemmatization** aims to obtain the canonical (grammatically correct) forms of individual words—the so-called **lemmas**. However, lemmatization is computationally more difficult and expensive compared to stemming and, in practice, it has been observed that stemming and lemmatization have little impact on the performance of text classification (*Influence of Word Normalization on Text Classification*, *Michal Toman*, *Roman Tesar*, and *Karel Jezek*, *Proceedings of InSciT*, pages 354–358, *2006*).

Before we jump into the next section, where we will train a machine learning model using the bag-of-words model, let's briefly talk about another useful topic called **stop-word removal**. Stop-words are simply those words that are extremely common in all sorts of texts and probably bear no (or only a little) useful information that can be used to distinguish between different classes of documents. Examples of stop-words are *is*, *and*, *has*, and *like*. Removing stop-words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which are already downweighting frequently occurring words.

In order to remove stop-words from the movie reviews, we will use the set of 127 English stop-words that is available from the NLTK library, which can be obtained by calling the `nltk.download` function:

```
>>> import nltk

>>> nltk.download('stopwords')
After we download the stop-words set, we can load and apply the
English stop-word set as follows:
>>> from nltk.corpus import stopwords

>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes'
...  ' running and runs a lot')[-10:]
...  if w not in stop]

['runner', 'like', 'run', 'run', 'lot']
```

Training a logistic regression model for document classification

In this section, we will train a logistic regression model to classify the movie reviews into *positive* and *negative* reviews based on the bag-of-words model. First, we will divide the `DataFrame` of cleaned text documents into 25,000 documents for training and 25,000 documents for testing:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Next, we will use a `GridSearchCV` object to find the optimal set of parameters for our logistic regression model using 5-fold stratified cross-validation:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer

>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...                'vect__stop_words': [stop, None],
...                'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter],
...                'clf__penalty': ['l1', 'l2'],
...                'clf__C': [1.0, 10.0, 100.0]},
...               {'vect__ngram_range': [(1,1)],
...                'vect__stop_words': [stop, None],
...                'vect__tokenizer': [tokenizer,
...                                    tokenizer_porter],
...                'vect__use_idf':[False],
...                'vect__norm':[None],
...                'clf__penalty': ['l1', 'l2'],
...                'clf__C': [1.0, 10.0, 100.0]}
...              ]
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                      ('clf',
...                       LogisticRegression(random_state=0,
...                                          solver='liblinear'))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...                            scoring='accuracy',
...                            cv=5, verbose=2,
...                            n_jobs=1)
```

```
>>> gs_lr_tfidf.fit(X_train, y_train)
```

Multiprocessing via the n_jobs parameter

Please note that it is highly recommended to set `n_jobs=-1` (instead of `n_jobs=1`) in the previous code example to utilize all available cores on your machine and speed up the grid search. However, some Windows users reported issues when running the previous code with the `n_jobs=-1` setting related to pickling the `tokenizer` and `tokenizer_porter` functions for multiprocessing on Windows. Another workaround would be to replace those two functions, `[tokenizer, tokenizer_porter]`, with `[str.split]`. However, note that replacement by the simple `str.split` would not support stemming.

When we initialized the `GridSearchCV` object and its parameter grid using the preceding code, we restricted ourselves to a limited number of parameter combinations, since the number of feature vectors, as well as the large vocabulary, can make the grid search computationally quite expensive. Using a standard desktop computer, our grid search may take up to 40 minutes to complete.

In the previous code example, we replaced `CountVectorizer` and `TfidfTransformer` from the previous subsection with `TfidfVectorizer`, which combines `CountVectorizer` with the `TfidfTransformer`. Our `param_grid` consisted of two parameter dictionaries. In the first dictionary, we used `TfidfVectorizer` with its default settings (`use_idf=True`, `smooth_idf=True`, and `norm='l2'`) to calculate the tf-idfs; in the second dictionary, we set those parameters to `use_idf=False`, `smooth_idf=False`, and `norm=None` in order to train a model based on raw term frequencies. Furthermore, for the logistic regression classifier itself, we trained models using L2 and L1 regularization via the penalty parameter and compared different regularization strengths by defining a range of values for the inverse-regularization parameter `C`.

After the grid search has finished, we can print the best parameter set:

```
>>> print('Best parameter set: %s ' % gs_lr_tfidf.best_params_)
Best parameter set: {'clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

As you can see in the preceding output, we obtained the best grid search results using the regular `tokenizer` without Porter stemming, no stop-word library, and tf-idfs in combination with a logistic regression classifier that uses L2-regularization with the regularization strength `C` of `10.0`.

Using the best model from this grid search, let's print the average 5-fold cross-validation accuracy scores on the training dataset and the classification accuracy on the test dataset:

```
>>> print('CV Accuracy: %.3f'
...       % gs_lr_tfidf.best_score_)
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Test Accuracy: %.3f'
...       % clf.score(X_test, y_test))
Test Accuracy: 0.899
```

The results reveal that our machine learning model can predict whether a movie review is positive or negative with 90 percent accuracy.

The naïve Bayes classifier

A still very popular classifier for text classification is the naïve Bayes classifier, which gained popularity in applications of email spam filtering. Naïve Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms. Although we don't discuss naïve Bayes classifiers in this book, the interested reader can find an article about naïve Bayes text classification that is freely available on arXiv (*Naive Bayes and Text Classification I – Introduction and Theory, S. Raschka, Computing Research Repository* (*CoRR*), abs/1410.5329, *2014*, `http://arxiv.org/pdf/1410.5329v3.pdf`).

Working with bigger data – online algorithms and out-of-core learning

If you executed the code examples in the previous section, you may have noticed that it could be computationally quite expensive to construct the feature vectors for the 50,000-movie review dataset during grid search. In many real-world applications, it is not uncommon to work with even larger datasets that can exceed our computer's memory. Since not everyone has access to supercomputer facilities, we will now apply a technique called **out-of-core learning**, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches of a dataset.

Text classification with recurrent neural networks

In *Chapter 16, Modeling Sequential Data Using Recurrent Neural Networks*, we will revisit this dataset and train a deep learning-based classifier (a recurrent neural network) to classify the reviews in the IMDb movie review dataset. This neural network-based classifier follows the same out-of-core principle using the stochastic gradient descent optimization algorithm but does not require the construction of a bag-of-words model.

Back in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, the concept of **stochastic gradient descent** was introduced; it is an optimization algorithm that updates the model's weights using one example at a time. In this section, we will make use of the `partial_fit` function of `SGDClassifier` in scikit-learn to stream the documents directly from our local drive and train a logistic regression model using small mini-batches of documents.

First, we will define a `tokenizer` function that cleans the unprocessed text data from the `movie_data.csv` file that we constructed at the beginning of this chapter and separate it into word tokens while removing stop-words:

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
...                   + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Next, we will define a generator function, `stream_docs`, that reads in and returns one document at a time:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # skip header
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

To verify that our `stream_docs` function works correctly, let's read in the first document from the `movie_data.csv` file, which should return a tuple consisting of the review text as well as the corresponding class label:

```
>>> next(stream_docs(path='movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

We will now define a function, `get_minibatch`, that will take a document stream from the `stream_docs` function and return a particular number of documents specified by the `size` parameter:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y
```

Unfortunately, we can't use `CountVectorizer` for out-of-core learning since it requires holding the complete vocabulary in memory. Also, `TfidfVectorizer` needs to keep all the feature vectors of the training dataset in memory to calculate the inverse document frequencies. However, another useful vectorizer for text processing implemented in scikit-learn is `HashingVectorizer`. `HashingVectorizer` is data-independent and makes use of the hashing trick via the 32-bit `MurmurHash3` function by Austin Appleby (`https://sites.google.com/site/murmurhash/`):

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1)
>>> doc_stream = stream_docs(path='movie_data.csv')
```

Using the preceding code, we initialized `HashingVectorizer` with our `tokenizer` function and set the number of features to `2**21`. Furthermore, we reinitialized a logistic regression classifier by setting the `loss` parameter of `SGDClassifier` to `'log'`. Note that by choosing a large number of features in `HashingVectorizer`, we reduce the chance of causing hash collisions, but we also increase the number of coefficients in our logistic regression model.

Now comes the really interesting part – having set up all the complementary functions, we can start the out-of-core learning using the following code:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                          100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:00:21
```

Again, we made use of the PyPrind package in order to estimate the progress of our learning algorithm. We initialized the progress bar object with 45 iterations and, in the following `for` loop, we iterated over 45 mini-batches of documents where each mini-batch consists of 1,000 documents. Having completed the incremental learning process, we will use the last 5,000 documents to evaluate the performance of our model:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Accuracy: %.3f' % clf.score(X_test, y_test))
Accuracy: 0.868
```

As you can see, the accuracy of the model is approximately 87 percent, slightly below the accuracy that we achieved in the previous section using the grid search for hyperparameter tuning. However, out-of-core learning is very memory efficient and it took less than a minute to complete. Finally, we can use the last 5,000 documents to update our model:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

The word2vec model

A more modern alternative to the bag-of-words model is **word2vec**, an algorithm that Google released in 2013 (*Efficient Estimation of Word Representations in Vector Space*, *T. Mikolov*, *K. Chen*, *G. Corrado*, and *J. Dean*, arXiv preprint arXiv:1301.3781, *2013*).

The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, *king* – *man* + *woman* = *queen*.

The original C-implementation with useful links to the relevant papers and alternative implementations can be found at `https://code.google.com/p/word2vec/`.

Topic modeling with Latent Dirichlet Allocation

Topic modeling describes the broad task of assigning topics to unlabeled text documents. For example, a typical application would be the categorization of documents in a large text corpus of newspaper articles. In applications of topic modeling, we then aim to assign category labels to those articles, for example, sports, finance, world news, politics, local news, and so forth. Thus, in the context of the broad categories of machine learning that we discussed in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, we can consider topic modeling as a clustering task, a subcategory of unsupervised learning.

In this section, we will discuss a popular technique for topic modeling called **Latent Dirichlet Allocation** (**LDA**). However, note that while Latent Dirichlet Allocation is often abbreviated as LDA, it is not to be confused with *linear discriminant analysis*, a supervised dimensionality reduction technique that was introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Embedding the movie review classifier into a web application

LDA is different from the supervised learning approach that we took in this chapter to classify movie reviews as positive and negative. Thus, if you are interested in embedding scikit-learn models into a web application via the Flask framework using the movie reviewer as an example, please feel free to jump to the next chapter and revisit this standalone section on topic modeling later on.

Decomposing text documents with LDA

Since the mathematics behind LDA is quite involved and requires knowledge about Bayesian inference, we will approach this topic from a practitioner's perspective and interpret LDA using layman's terms. However, the interested reader can read more about LDA in the following research paper: *Latent Dirichlet Allocation*, *David M. Blei*, *Andrew Y. Ng*, and *Michael I. Jordan*, *Journal of Machine Learning Research 3*, pages: 993-1022, *Jan 2003*.

LDA is a generative probabilistic model that tries to find groups of words that appear frequently together across different documents. These frequently appearing words represent our topics, assuming that each document is a mixture of different words. The input to an LDA is the bag-of-words model that we discussed earlier in this chapter. Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:

- A document-to-topic matrix
- A word-to-topic matrix

LDA decomposes the bag-of-words matrix in such a way that if we multiply those two matrices together, we will be able to reproduce the input, the bag-of-words matrix, with the lowest possible error. In practice, we are interested in those topics that LDA found in the bag-of-words matrix. The only downside may be that we must define the number of topics beforehand—the number of topics is a hyperparameter of LDA that has to be specified manually.

LDA with scikit-learn

In this subsection, we will use the `LatentDirichletAllocation` class implemented in scikit-learn to decompose the movie review dataset and categorize it into different topics. In the following example, we will restrict the analysis to 10 different topics, but readers are encouraged to experiment with the hyperparameters of the algorithm to further explore the topics that can be found in this dataset.

First, we are going to load the dataset into a pandas `DataFrame` using the local `movie_data.csv` file of the movie reviews that we created at the beginning of this chapter:

```
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Next, we are going to use the already familiar `CountVectorizer` to create the bag-of-words matrix as input to the LDA.

For convenience, we will use scikit-learn's built-in English stop-word library via `stop_words='english'`:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                         max_df=.1,
...                         max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Notice that we set the maximum document frequency of words to be considered to 10 percent (`max_df=.1`) to exclude words that occur too frequently across documents. The rationale behind the removal of frequently occurring words is that these might be common words appearing across all documents that are, therefore, less likely to be associated with a specific topic category of a given document. Also, we limited the number of words to be considered to the most frequently occurring 5,000 words (`max_features=5000`), to limit the dimensionality of this dataset to improve the inference performed by LDA. However, both `max_df=.1` and `max_features=5000` are hyperparameter values chosen arbitrarily, and readers are encouraged to tune them while comparing the results.

The following code example demonstrates how to fit a `LatentDirichletAllocation` estimator to the bag-of-words matrix and infer the 10 different topics from the documents (note that the model fitting can take up to 5 minutes or more on a laptop or standard desktop computer):

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> lda = LatentDirichletAllocation(n_components=10,
...                                 random_state=123,
...                                 learning_method='batch')
>>> X_topics = lda.fit_transform(X)
```

By setting `learning_method='batch'`, we let the `lda` estimator do its estimation based on all available training data (the bag-of-words matrix) in one iteration, which is slower than the alternative `'online'` learning method but can lead to more accurate results (setting `learning_method='online'` is analogous to online or mini-batch learning, which we discussed in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, and in this chapter).

Expectation-Maximization

The scikit-learn library's implementation of LDA uses the **expectation-maximization** (**EM**) algorithm to update its parameter estimates iteratively. We haven't discussed the EM algorithm in this chapter, but if you are curious to learn more, please see the excellent overview on Wikipedia (https://en.wikipedia.org/wiki/Expectation-maximization_algorithm) and the detailed tutorial on how it is used in LDA in Colorado Reed's tutorial, *Latent Dirichlet Allocation: Towards a Deeper Understanding*, which is freely available at http://obphio.us/pdfs/lda_tutorial.pdf.

After fitting the LDA, we now have access to the components_ attribute of the lda instance, which stores a matrix containing the word importance (here, 5000) for each of the 10 topics in increasing order:

```
>>> lda.components_.shape
(10, 5000)
```

To analyze the results, let's print the five most important words for each of the 10 topics. Note that the word importance values are ranked in increasing order. Thus, to print the top five words, we need to sort the topic array in reverse order:

```
>>> n_top_words = 5
>>> feature_names = count.get_feature_names()
>>> for topic_idx, topic in enumerate(lda.components_):
...     print("Topic %d:" % (topic_idx + 1))
...     print(" ".join([feature_names[i]
...                     for i in topic.argsort()\
...                     [:-n_top_words - 1:-1]]))

Topic 1:
worst minutes awful script stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
```

```
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool
```

Based on reading the five most important words for each topic, you may guess that the LDA identified the following topics:

1. Generally bad movies (not really a topic category)
2. Movies about families
3. War movies
4. Art movies
5. Crime movies
6. Horror movies
7. Comedy movies reviews
8. Movies somehow related to TV shows
9. Movies based on books
10. Action movies

To confirm that the categories make sense based on the reviews, let's plot three movies from the horror movie category (horror movies belong to category 6 at index position `5`):

```
>>> horror = X_topics[:, 5].argsort()[::-1]
>>> for iter_idx, movie_idx in enumerate(horror[:3]):
...     print('\nHorror movie #%d:' % (iter_idx + 1))
...     print(df['review'][movie_idx][:300], '...')
Horror movie #1:
House of Dracula works from the same basic premise as House of
Frankenstein from the year before; namely that Universal's three most
famous monsters; Dracula, Frankenstein's Monster and The Wolf Man are
appearing in the movie together. Naturally, the film is rather messy
therefore, but the fact that ...

Horror movie #2:
Okay, what the hell kind of TRASH have I been watching now? "The
Witches' Mountain" has got to be one of the most incoherent and insane
Spanish exploitation flicks ever and yet, at the same time, it's also
strangely compelling. There's absolutely nothing that makes sense here
and I even doubt there ...
```

```
Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a
total freakfest from start to finish. A fun freakfest at that, but at
times it was a tad too reliant on kitsch rather than the horror. The
story is difficult to summarize succinctly: a carefree, normal teenage
girl starts coming fac ...
```

Using the preceding code example, we printed the first 300 characters from the top three horror movies. The reviews—even though we don't know which exact movie they belong to—sound like reviews of horror movies (however, one might argue that `Horror movie #2` could also be a good fit for topic category 1: *Generally bad movies*).

Summary

In this chapter, you learned how to use machine learning algorithms to classify text documents based on their polarity, which is a basic task in sentiment analysis in the field of NLP. Not only did you learn how to encode a document as a feature vector using the bag-of-words model, but you also learned how to weight the term frequency by relevance using tf-idf.

Working with text data can be computationally quite expensive due to the large feature vectors that are created during this process; in the last section, we covered how to utilize out-of-core or incremental learning to train a machine learning algorithm without loading the whole dataset into a computer's memory.

Lastly, you were introduced to the concept of topic modeling using LDA to categorize the movie reviews into different categories in an unsupervised fashion.

In the next chapter, we will use our document classifier and learn how to embed it into a web application.

9

Embedding a Machine Learning Model into a Web Application

In the previous chapters, you learned about the many different machine learning concepts and algorithms that can help us with better and more efficient decision-making. However, machine learning techniques are not limited to offline applications and analyses, and they have become the predictive engine of various web services. For example, popular and useful applications of machine learning models in web applications include spam detection in submission forms, search engines, recommendation systems for media or shopping portals, and many more.

In this chapter, you will learn how to embed a machine learning model into a web application that can not only classify, but also learn from data in real time. The topics that we will cover are as follows:

- Saving the current state of a trained machine learning model
- Using SQLite databases for data storage
- Developing a web application using the popular Flask web framework
- Deploying a machine learning application to a public web server

Serializing fitted scikit-learn estimators

Training a machine learning model can be computationally expensive, as you saw in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*. Surely, we don't want to retrain our model every time we close our Python interpreter and want to make a new prediction or reload our web application?

One option for model persistence is Python's in-built `pickle` module (`https://docs.python.org/3.7/library/pickle.html`), which allows us to serialize and deserialize Python object structures to compact bytecode so that we can save our classifier in its current state and reload it if we want to classify new, unlabeled examples, without needing the model to learn from the training data all over again. Before you execute the following code, please make sure that you have trained the out-of-core logistic regression model from the last section of *Chapter 8* and have it ready in your current Python session:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...             open(os.path.join(dest, 'stopwords.pkl'), 'wb'),
...             protocol=4)
>>> pickle.dump(clf,
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...             protocol=4)
```

Using the preceding code, we created a `movieclassifier` directory where we will later store the files and data for our web application. Within this `movieclassifier` directory, we created a `pkl_objects` subdirectory to save the serialized Python objects to our local hard drive or solid-state drive. Via the `dump` method of the `pickle` module, we then serialized the trained logistic regression model as well as the stop-word set from the **Natural Language Toolkit** (**NLTK**) library, so that we don't have to install the NLTK vocabulary on our server.

The `dump` method takes as its first argument the object that we want to pickle. For the second argument, we provided an open file object that the Python object will be written to. Via the `wb` argument inside the `open` function, we opened the file in binary mode for pickle, and we set `protocol=4` to choose the latest and most efficient pickle protocol that was added to Python 3.4, which is compatible with Python 3.4 or newer. If you have problems using `protocol=4`, please check whether you are using the latest Python 3 version—Python 3.7 is recommended for this book. Alternatively, you may consider choosing a lower protocol number.

Also note that if you are using a custom web server, you have to ensure that the Python installation on that server is compatible with this protocol version as well.

Serializing NumPy arrays with joblib

Our logistic regression model contains several NumPy arrays, such as the weight vector, and a more efficient way to serialize NumPy arrays is to use the alternative `joblib` library. To ensure compatibility with the server environment that we will use in later sections, we will use the standard pickle approach. If you are interested, you can find more information about `joblib` at `https://joblib.readthedocs.io`.

We don't need to pickle `HashingVectorizer`, since it does not need to be fitted. Instead, we can create a new Python script file from which we can import the vectorizer into our current Python session. Now, copy the following code and save it as `vectorizer.py` in the `movieclassifier` directory:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(os.path.join(
                   cur_dir, 'pkl_objects', 'stopwords.pkl'),
                   'rb'))

def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
                  + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

After we have pickled the Python objects and created the `vectorizer.py` file, it would be a good idea to restart our Python interpreter or Jupyter Notebook kernel to test whether we can deserialize the objects without error.

Pickle can be a security risk

Please note that unpickling data from an untrusted source can be a potential security risk, since the `pickle` module is not secured against malicious code. Since `pickle` was designed to serialize arbitrary objects, the unpickling process will execute code that has been stored in a pickle file. Thus, if you receive pickle files from an untrusted source (for example, by downloading them from the internet), please proceed with extra care and unpickle the items in a virtual environment and/or on a non-essential machine that does not store important data that no one except you should have access to.

From your terminal, navigate to the `movieclassifier` directory, start a new Python session, and execute the following code to verify that you can import the `vectorizer` and unpickle the classifier:

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
>>> clf = pickle.load(open(os.path.join(
...                     'pkl_objects', 'classifier.pkl'),
...                     'rb'))
```

After we have successfully loaded the `vectorizer` and unpickled the classifier, we can use these objects to preprocess document examples and make predictions about their sentiments:

```
>>> import numpy as np
>>> label = {0:'negative', 1:'positive'}

>>> example = ["I love this movie. It's amazing."]
>>> X = vect.transform(example)
>>> print('Prediction: %s\nProbability: %.2f%%' %\
...       (label[clf.predict(X)[0]],
...        np.max(clf.predict_proba(X))*100))
Prediction: positive
Probability: 95.55%
```

Since our classifier returns the class label predictions as integers, we defined a simple Python dictionary to map these integers to their sentiment (`"positive"` or `"negative"`). While this is a simple application with two classes only, it should be noted that this dictionary-mapping approach also generalizes to multiclass settings. Furthermore, this mapping dictionary should also be archived alongside the model.

In this case, since the dictionary definition only consists of one line of code, we will not go to the trouble of serializing it using pickle. However, in real-world applications with more extensive mapping dictionaries, you can utilize the same pickle.dump and pickle.load commands that we used in the previous code example.

Continuing with the discussion of the previous code example, we then used HashingVectorizer to transform the simple example document into a word vector, X. Finally, we used the predict method of the logistic regression classifier to predict the class label, as well as the predict_proba method to return the corresponding probability of our prediction. Note that the predict_proba method call returns an array with a probability value for each unique class label. Since the class label with the largest probability corresponds to the class label that is returned by the predict call, we used the np.max function to return the probability of the predicted class.

Setting up an SQLite database for data storage

In this section, we will set up a simple SQLite database to collect optional feedback about the predictions from users of the web application. We can use this feedback to update our classification model. SQLite is an open source SQL database engine that doesn't require a separate server to operate, which makes it ideal for smaller projects and simple web applications. Essentially, an SQLite database can be understood as a single, self-contained database file that allows us to directly access storage files.

Furthermore, SQLite doesn't require any system-specific configuration and is supported by all common operating systems. It has gained a reputation for being very reliable and is used by popular companies such as Google, Mozilla, Adobe, Apple, Microsoft, and many more. If you want to learn more about SQLite, visit the official website at http://www.sqlite.org.

Fortunately, following Python's *batteries included* philosophy, there is already an API in the Python standard library, sqlite3, which allows us to work with SQLite databases. (For more information about sqlite3, please visit https://docs.python.org/3.7/library/sqlite3.html.)

By executing the following code, we will create a new SQLite database inside the movieclassifier directory and store two example movie reviews:

```
>>> import sqlite3
>>> import os

>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('DROP TABLE IF EXISTS review_db')
>>> c.execute('CREATE TABLE review_db'\
...           ' (review TEXT, sentiment INTEGER, date TEXT)')

>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example1, 1))

>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db"\
...           " (review, sentiment, date) VALUES"\
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()
```

Following the preceding code example, we created a connection (`conn`) to an SQLite database file by calling the `connect` method of the `sqlite3` library, which created the new database file `reviews.sqlite` in the `movieclassifier` directory if it didn't already exist.

Next, we created a cursor via the `cursor` method, which allows us to traverse over the database records using the versatile SQL syntax. Via the first `execute` call, we then created a new database table, `review_db`. We used this to store and access database entries. Along with `review_db`, we also created three columns in this database table: `review`, `sentiment`, and `date`. We used these to store two example movie reviews and respective class labels (sentiments).

Using the `DATETIME('now')` SQL command, we also added date and timestamps to our entries. In addition to the timestamps, we used the question mark symbols (`?`) to pass the movie review texts (`example1` and `example2`) and the corresponding class labels (`1` and `0`) as positional arguments to the `execute` method, as members of a tuple. Lastly, we called the `commit` method to save the changes that we made to the database and closed the connection via the `close` method.

To check if the entries have been stored in the database table correctly, we will now reopen the connection to the database and use the SQL `SELECT` command to fetch all rows in the database table that have been committed between the beginning of the year 2017 and today:

```
>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date"\
...           " BETWEEN '2017-01-01 00:00:00' AND DATETIME('now')")
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2019-06-15 17:53:46'), ('I disliked this
movie', 0, '2019-06-15 17:53:46')]
```

Alternatively, we could also use the free DB browser for SQLite app (available at https://sqlitebrowser.org/dl/), which offers a nice graphical user interface for working with SQLite databases, as shown in the following figure:

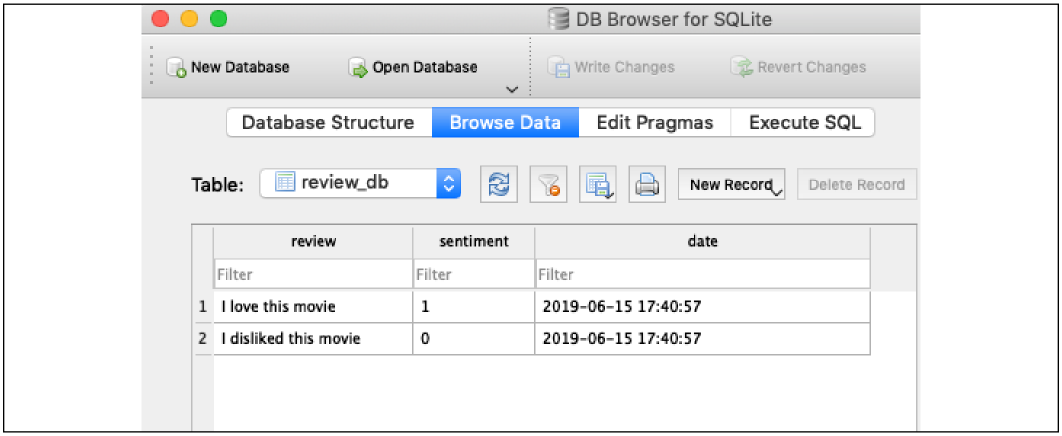


Table: review_db

	review	sentiment	date
1	I love this movie	1	2019-06-15 17:40:57
2	I disliked this movie	0	2019-06-15 17:40:57

Developing a web application with Flask

Having prepared the code for classifying movie reviews in the previous subsection, let's discuss the basics of the Flask web framework to develop our web application. Since Armin Ronacher's initial release of Flask in 2010, the framework has gained huge popularity, and examples of popular applications that use Flask include LinkedIn and Pinterest. Since Flask is written in Python, it provides us Python programmers with a convenient interface for embedding existing Python code, such as our movie classifier.

The Flask microframework

Flask is also known as a **microframework**, which means that its core is kept lean and simple but it can be easily extended with other libraries. Although the learning curve of the lightweight Flask API is not nearly as steep as those of other popular Python web frameworks, such as Django, you are encouraged to take a look at the official Flask documentation at https://flask.palletsprojects.com/en/1.0.x/ to learn more about its functionality.

If the Flask library is not already installed in your current Python environment, you can simply install it via conda or pip from your terminal (at the time of writing, the latest stable release was version 1.0.2):

```
conda install flask
# or: pip install flask
```

Our first Flask web application

In this subsection, we will develop a very simple web application to become more familiar with the Flask API before we implement our movie classifier. This first application that we are going to build consists of a simple web page with a form field that lets us enter a name. After submitting the name to the web application, it will render it on a new page. While this is a very simple example of a web application, it helps with building an understanding of how to store and pass variables and values between the different parts of our code within the Flask framework.

First, we create a directory tree:

```
1st_flask_app_1/
    app.py
    templates/
        first_app.html
```

The app.py file will contain the main code that will be executed by the Python interpreter to run the Flask web application. The templates directory is the directory in which Flask will look for static HTML files for rendering in the web browser. Let's now take a look at the contents of app.py:

```
from flask import Flask, render_template

app = Flask(__name__)
@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

After looking at the previous code example, let's discuss the individual pieces step by step:

1. We ran our application as a single module; thus, we initialized a new Flask instance with the argument __name__ to let Flask know that it can find the HTML template folder (templates) in the same directory where it is located.

2. Next, we used the route decorator (`@app.route('/')`) to specify the URL that should trigger the execution of the `index` function.
3. Here, our `index` function simply rendered the `first_app.html` HTML file, which is located in the `templates` folder.
4. Lastly, we used the `run` function to run the application on the server only when this script was directly executed by the Python interpreter, which we ensured using the `if` statement with `__name__ == '__main__'`.

Now, let's take a look at the contents of the `first_app.html` file:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
    <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```

HTML basics

If you are not familiar with the HTML syntax yet, visit `https://developer.mozilla.org/en-US/docs/Web/HTML` for useful tutorials on learning the basics of HTML.

Here, we have simply filled an empty HTML template file with a `<div>` element (a block-level element) that contains this sentence: `Hi, this is my first Flask web app!`.

Conveniently, Flask allows us to run our applications locally, which is useful for developing and testing web applications before we deploy them on a public web server. Now, let's start our web application by executing the command from the terminal inside the `1st_flask_app_1` directory:

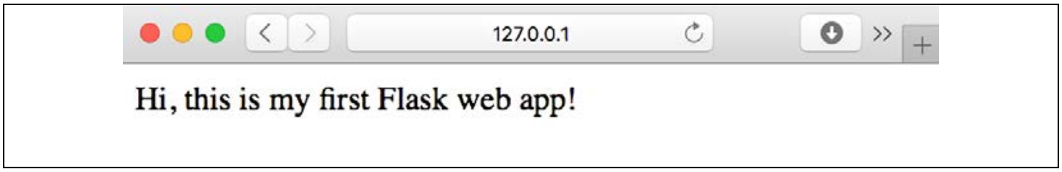
```
python3 app.py
```

We should see a line such as the following displayed in the terminal:

```
* Running on http://127.0.0.1:5000/
```

This line contains the address of our local server. We can enter this address in our web browser to see the web application in action.

If everything has executed correctly, we should see a simple website with the content `Hi, this is my first Flask web app!` as shown in the following figure:



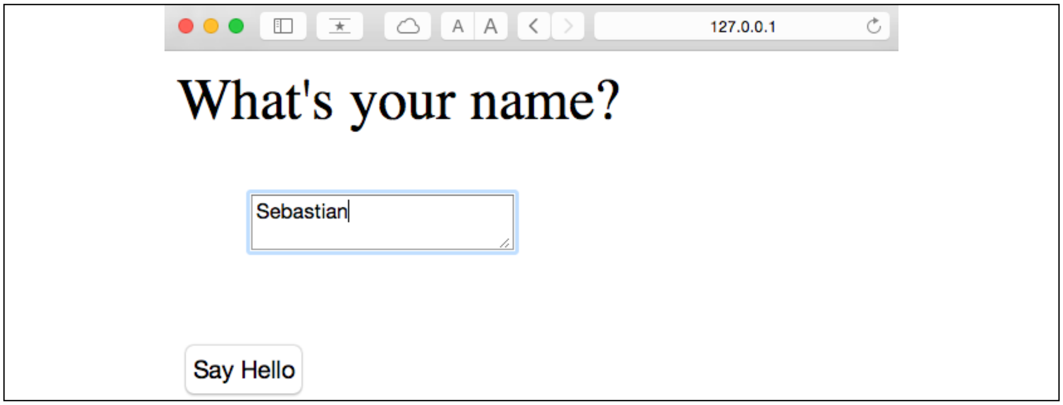
Hi, this is my first Flask web app!

Form validation and rendering

In this subsection, we will extend our simple Flask web application with HTML form elements to learn how to collect data from a user using the WTForms library (`https://wtforms.readthedocs.org/en/latest/`), which can be installed via `conda` or `pip`:

```
conda install wtforms
# or pip install wtforms
```

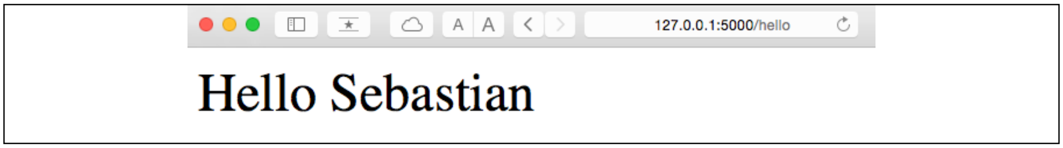
This web application will prompt a user to type in his or her name into a text field, as shown in the following screenshot:



What's your name?

Sebastian

After the submission button (**Say Hello**) has been clicked and the form has been validated, a new HTML page will be rendered to display the user's name:



Hello Sebastian

Setting up the directory structure

The new directory structure that we need to set up for this application looks like this:

```
1st_flask_app_2/
    app.py
    static/
        style.css
    templates/
        _formhelpers.html
        first_app.html
        hello.html
```

The following are the contents of our modified `app.py` file:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)
if __name__ == '__main__':
    app.run(debug=True)
```

Let's discuss what the previous code does step by step:

1. Using `wtforms`, we extended the `index` function with a text field that we will embed in our start page using the `TextAreaField` class, which automatically checks whether a user has provided valid input text or not.
2. Furthermore, we defined a new function, `hello`, which will render an HTML page, `hello.html`, after validating the HTML form.

3. Here, we used the POST method to transport the form data to the server in the message body. Finally, by setting the debug=True argument inside the app.run method, we further activated Flask's debugger. This is a useful feature for developing new web applications.

Implementing a macro using the Jinja2 templating engine

Now, we will implement a generic macro in the _formhelpers.html file via the Jinja2 templating engine, which we will later import in our first_app.html file to render the text field:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
  </dt>
{% endmacro %}
```

An in-depth discussion about the Jinja2 templating language is beyond the scope of this book. However, you can find comprehensive documentation on the Jinja2 syntax at http://jinja.pocoo.org.

Adding style via CSS

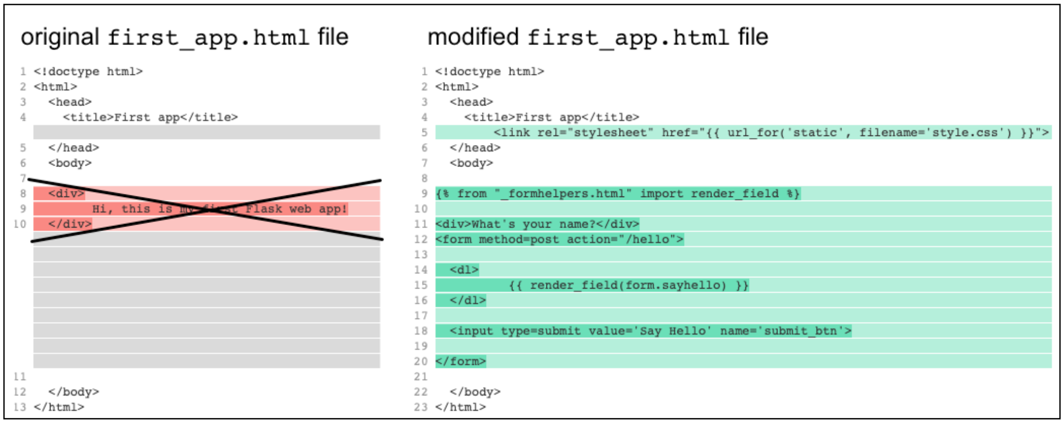
Next, we will set up a simple **Cascading Style Sheets** (**CSS**) file, style.css, to demonstrate how the look and feel of HTML documents can be modified. We have to save the following CSS file, which will simply double the font size of our HTML body elements, in a subdirectory called static, which is the default directory where Flask looks for static files such as CSS. The file content is as follows:

```
body {
     font-size: 2em;
}
```

The following are the contents of the modified `first_app.html` file that will now render a text form where a user can enter a name:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
      <link rel="stylesheet"
       href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>
    {% from "_formhelpers.html" import render_field %}
    <div>What's your name?</div>
    <form method=post action="/hello">
      <dl>
        {{ render_field(form.sayhello) }}
      </dl>
      <input type=submit value='Say Hello' name='submit_btn'>
    </form>
  </body>
</html>
```

In the header section of `first_app.html`, we loaded the CSS file. It should now alter the size of all text elements in the HTML body. In the HTML body section, we imported the form macro from `_formhelpers.html`, and we rendered the `sayhello` form that we specified in the `app.py` file. Furthermore, we added a button to the same form element so that a user can submit the text field entry. The changes between the original and modified `first_app.html` file are illustrated in the following figure:



original `first_app.html` file

```
<!doctype html>
<html>
  <head>
    <title>First app</title>

  </head>
  <body>

  <div>
        Hi, this is my first Flask web app!
  </div>


  </body>
</html>
```

modified `first_app.html` file

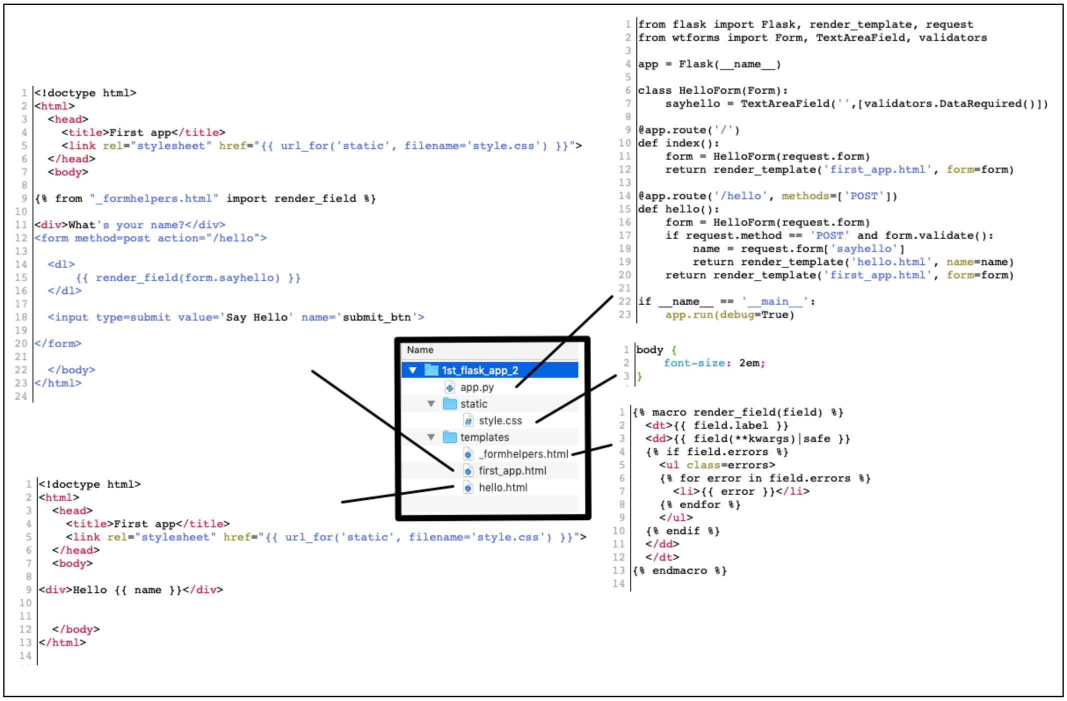
```
<!doctype html>
<html>
  <head>
    <title>First app</title>
        <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

{% from "_formhelpers.html" import render_field %}

<div>What's your name?</div>
<form method=post action="/hello">

  <dl>
          {{ render_field(form.sayhello) }}
  </dl>

  <input type=submit value='Say Hello' name='submit_btn'>

</form>

  </body>
</html>
```

Creating the result page

Lastly, we will create a `hello.html` file that will be rendered via the `render_template('hello.html', name=name)` line return inside the `hello` function, which we defined in the `app.py` script to display the text that a user submitted via the text field. The file content is as follows:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
      <link rel="stylesheet"
       href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>
    <div>Hello {{ name }}</div>
  </body>

</html>
```

Since we have covered a lot of ground in the previous section, the following figure provides an overview of the files we have created:



```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

{% from "_formhelpers.html" import render_field %}

<div>What's your name?</div>
<form method=post action="/hello">

  <dl>
      {{ render_field(form.sayhello) }}
  </dl>

  <input type=submit value='Say Hello' name='submit_btn'>

</form>

  </body>
</html>
```

```
body {
    font-size: 2em;
}
```

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
  </dt>
{% endmacro %}
```

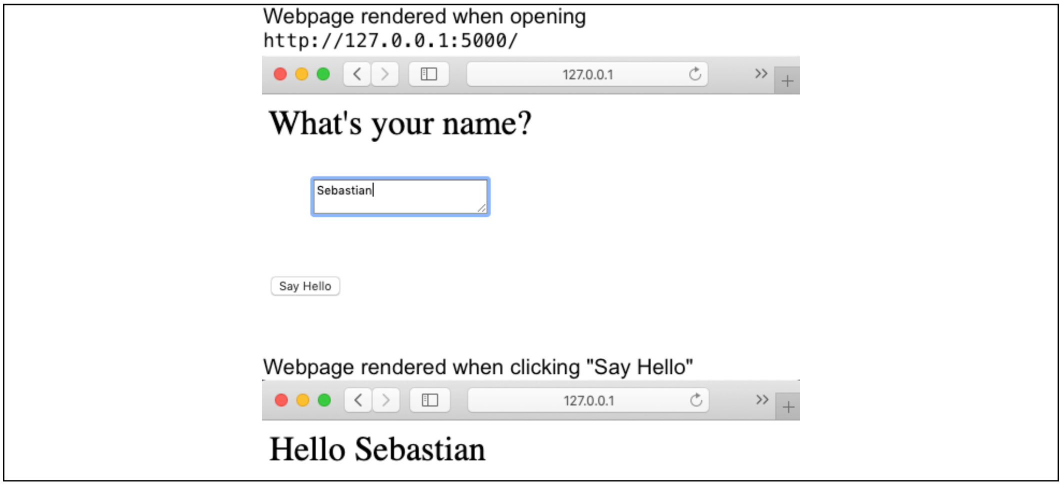
```
<!doctype html>
<html>
  <head>
    <title>First app</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

<div>Hello {{ name }}</div>


  </body>
</html>
```

Please note that you do not need to copy any code from the previous figure, since all file contents are present in the previous sections. For your convenience, copies of all files can also be found online at https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch09/1st_flask_app_2.

Having set up our modified Flask web application, we can run it locally by executing the following command from the application's main directory:

```
python3 app.py
```

Then, to see the resulting web page, enter the IP address shown in your terminal, which is usually http://127.0.0.1:5000/, into your web browser to view the rendered web app as summarized in the following figure:


Webpage rendered when opening
http://127.0.0.1:5000/
127.0.0.1

What's your name?

Sebastian

Say Hello
Webpage rendered when clicking "Say Hello"
127.0.0.1

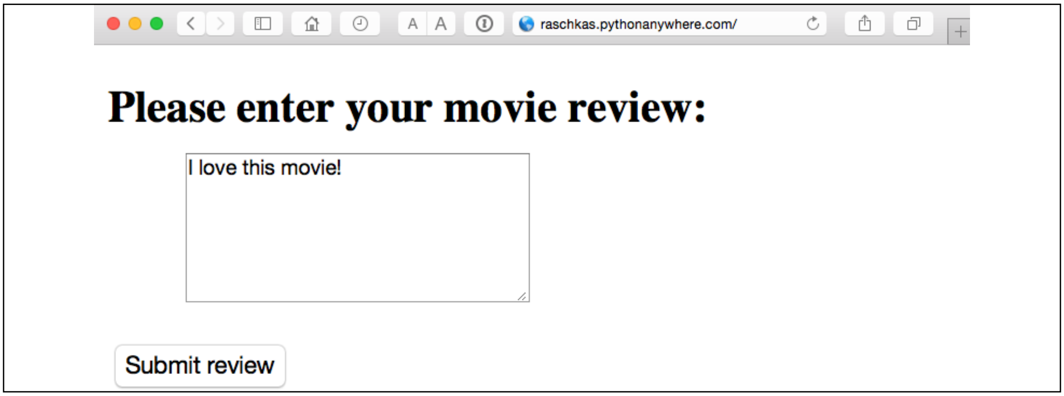
Hello Sebastian

Flask documentation and examples

If you are new to web development, some of those concepts may seem very complicated at first sight. In that case, simply set up the preceding files in a directory on your hard drive and examine them closely. You will see that the Flask web framework is relatively straightforward and much simpler than it might initially appear! Also, for more help, don't forget to consult the excellent Flask documentation and examples at http://flask.pocoo.org/docs/1.0/.

Turning the movie review classifier into a web application

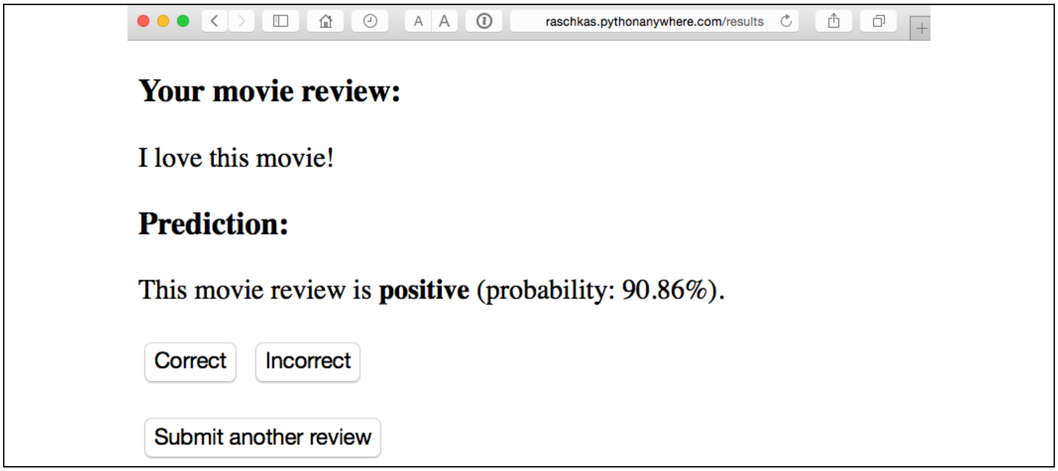
Now that we are somewhat familiar with the basics of Flask web development, let's advance to the next step and implement our movie classifier into a web application. In this section, we will develop a web application that will first prompt a user to enter a movie review, as shown in the following screenshot:



Please enter your movie review:

I love this movie!

After the review has been submitted, the user will see a new page that shows the predicted class label and the probability of the prediction. Furthermore, the user will be able to provide feedback about this prediction by clicking on the **Correct** or **Incorrect** button, as shown in the following screenshot:



Your movie review:

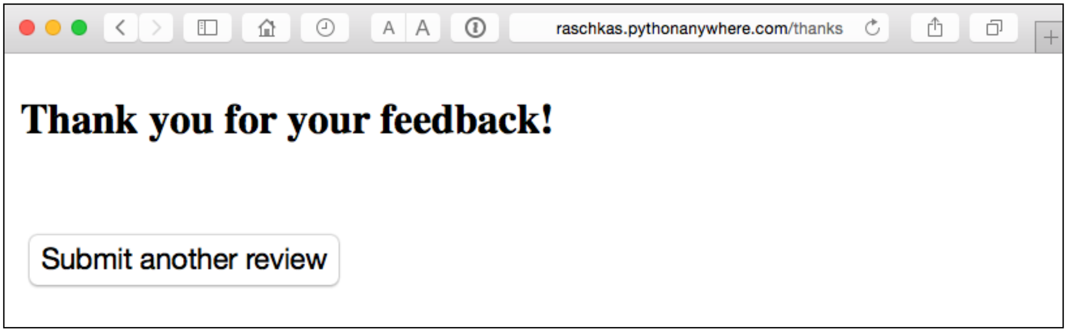
I love this movie!

Prediction:

This movie review is **positive** (probability: 90.86%).

If a user clicked on either the **Correct** or **Incorrect** button, our classification model will be updated with respect to the user's feedback. Furthermore, we will also store the movie review text provided by the user, as well as the suggested class label, which can be inferred from the button click, in an SQLite database for future reference. (Alternatively, a user could skip the update step and click the **Submit another review** button to submit another review.)

The third page that the user will see after clicking on one of the feedback buttons is a simple *thank you* screen with a **Submit another review** button that redirects the user back to the start page. This is shown in the following screenshot:



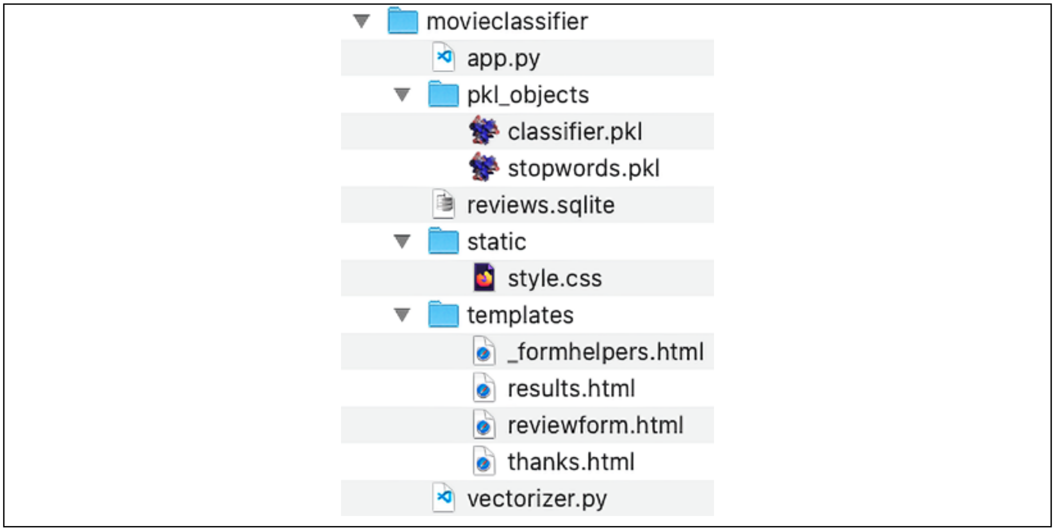
Thank you for your feedback!

Live demo

Before we take a closer look at the code implementation of this web application, take a look at this live demo at `http://raschkas.pythonanywhere.com` to get a better understanding of what we are trying to accomplish in this section.

Files and folders – looking at the directory tree

To start with the big picture, let's take a look at the directory tree that we are going to create for this movie classification application, which is shown here:



```
▼ movieclassifier
    app.py
  ▼ pkl_objects
      classifier.pkl
      stopwords.pkl
    reviews.sqlite
  ▼ static
      style.css
  ▼ templates
      _formhelpers.html
      results.html
      reviewform.html
      thanks.html
    vectorizer.py
```

Earlier in this chapter, we created the `vectorizer.py` file, the SQLite database, `reviews.sqlite`, and the `pkl_objects` subdirectory with the pickled Python objects.

The `app.py` file in the main directory is the Python script that contains our Flask code, and we will use the `review.sqlite` database file (which we created earlier in this chapter) to store the movie reviews that are being submitted to our web application. The `templates` subdirectory contains the HTML templates that will be rendered by Flask and displayed in the browser, and the `static` subdirectory will contain a simple CSS file to adjust the look of the rendered HTML code.

Getting the movieclassifier code files

A separate directory containing the movie review classifier application with the code discussed in this section is provided with the code examples for this book, which you can either obtain directly from Packt or download from GitHub at `https://github.com/rasbt/python-machine-learning-book-3rd-edition/`. The code in this section can be found in the `.../code/ch09/movieclassifier` subdirectory.

Implementing the main application as app.py

Since the `app.py` file is rather long, we will conquer it in two steps. The first section of `app.py` imports the Python modules and objects that we are going to need, as well as the code to unpickle and set up our classification model:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np

# import HashingVectorizer from local dir
from vectorizer import vect

app = Flask(__name__)

######## Preparing the Classifier
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                  'pkl_objects', 'classifier.pkl'),
                  'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = np.max(clf.predict_proba(X))
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
              " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

This first part of the `app.py` script should look very familiar by now. We simply imported `HashingVectorizer` and unpickled the logistic regression classifier. Next, we defined a `classify` function to return the predicted class label, as well as the corresponding probability prediction of a given text document. The `train` function can be used to update the classifier, given that a document and a class label are provided.

Using the `sqlite_entry` function, we can store a submitted movie review in our SQLite database along with its class label and timestamp for our personal records. Note that the `clf` object will be reset to its original, pickled state if we restart the web application. At the end of this chapter, you will learn how to use the data that we collect in the SQLite database to update the classifier permanently.

The concepts in the second part of the `app.py` script should also look quite familiar:

```
######## Flask
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                                 validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                               content=review,
                               prediction=y,
                               probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

We defined a ReviewForm class that instantiates a TextAreaField, which will be rendered in the reviewform.html template file (the landing page of our web application). This, in turn, will be rendered by the index function. With the validators.length(min=15) parameter, we require the user to enter a review that contains at least 15 characters. Inside the results function, we fetch the contents of the submitted web form and pass it on to our classifier to predict the sentiment of the movie classifier, which will then be displayed in the rendered results.html template.

The feedback function, which we implemented in app.py in the previous subsection, may look a little bit complicated at first glance. It essentially fetches the predicted class label from the results.html template if a user clicked on the **Correct** or **Incorrect** feedback button, and it transforms the predicted sentiment back into an integer class label that will be used to update the classifier via the train function, which we implemented in the first section of the app.py script. Also, a new entry to the SQLite database will be made via the sqlite_entry function if feedback was provided, and eventually, the thanks.html template will be rendered to thank the user for the feedback.

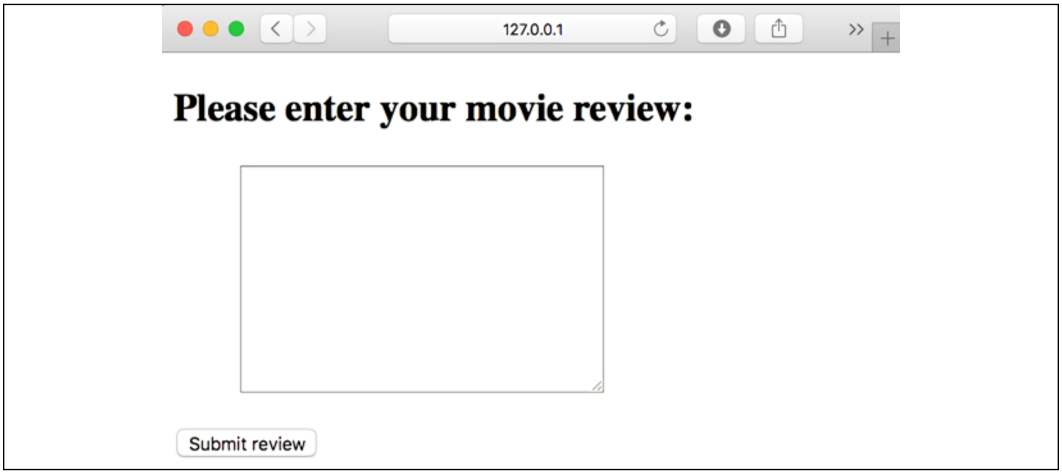
Setting up the review form

Next, let's take a look at the reviewform.html template, which constitutes the starting page of our application:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
      <link rel="stylesheet"
       href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

    <h2>Please enter your movie review:</h2>

    {% from "_formhelpers.html" import render_field %}

    <form method=post action="/results">
      <dl>
        {{ render_field(form.moviereview, cols='30', rows='10') }}
      </dl>
```

```
          <div>
            <input type=submit value='Submit review' name='submit_btn'>
          </div>
        </form>

      </body>
    </html>
```

Here, we simply imported the same `_formhelpers.html` template that we defined in the *Form validation and rendering* section earlier in this chapter. The `render_field` function of this macro is used to render a `TextAreaField` where a user can provide a movie review and submit it via the **Submit review** button displayed at the bottom of the page. This `TextAreaField` is 30 columns wide and 10 rows tall, and will look like this:



Please enter your movie review:

Creating a results page template

Our next template, `results.html`, looks a little bit more interesting:

```
    <!doctype html>
    <html>
      <head>
        <title>Movie Classification</title>
          <link rel="stylesheet"
           href="{{ url_for('static', filename='style.css') }}">
      </head>
      <body>

        <h3>Your movie review:</h3>
        <div>{{ content }}</div>

        <h3>Prediction:</h3>
        <div>This movie review is <strong>{{ prediction }}</strong>
        (probability: {{ probability }}%).</div>

        <div id='button'>
          <form action="/thanks" method="post">
            <input type=submit value='Correct' name='feedback_button'>
            <input type=submit value='Incorrect' name='feedback_button'>
            <input type=hidden value='{{ prediction }}' name='prediction'>
            <input type=hidden value='{{ content }}' name='review'>
          </form>
        </div>

        <div id='button'>
          <form action="/">
            <input type=submit value='Submit another review'>
          </form>
        </div>

      </body>
    </html>
```

First, we inserted the submitted review, as well as the results of the prediction, in the corresponding fields `{{ content }}`, `{{ prediction }}`, and `{{ probability }}`. You may notice that we used the `{{ content }}` and `{{ prediction }}` placeholder variables (in this context, also known as *hidden fields*) a second time in the form that contains the **Correct** and **Incorrect** buttons. This is a workaround to `POST` those values back to the server to update the classifier and store the review in case the user clicks on one of those two buttons.

Furthermore, we imported a CSS file (`style.css`) at the beginning of the `results.html` file. The setup of this file is quite simple: it limits the width of the contents of this web application to 600 pixels and moves the **Incorrect** and **Correct** buttons labeled with the div id `button` down by 20 pixels:

```
  body{
    width:600px;
  }

  .button{
    padding-top: 20px;
  }
```

This CSS file is merely a placeholder, so please feel free to modify it to adjust the look and feel of the web application to your liking.

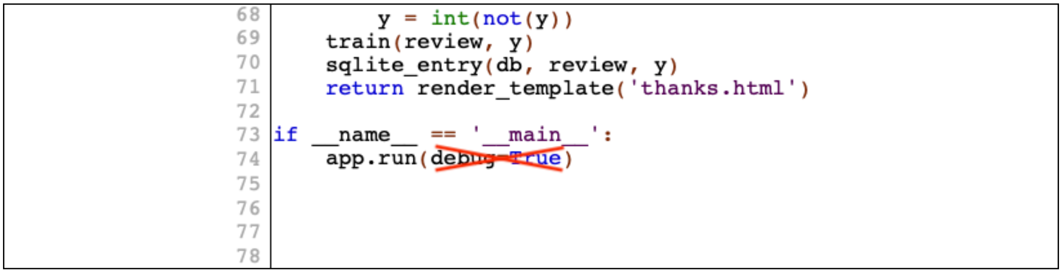
The last HTML file we will implement for our web application is the `thanks.html` template. As the name suggests, it simply provides a nice *thank you* message to the user after providing feedback via the **Correct** or **Incorrect** button. Furthermore, we will put a **Submit another review** button at the bottom of this page, which will redirect the user to the starting page. The contents of the `thanks.html` file are as follows:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
      <link rel="stylesheet"
       href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

    <h3>Thank you for your feedback!</h3>

    <div id='button'>
      <form action="/">
        <input type=submit value='Submit another review'>
      </form>
    </div>

  </body>
</html>
```

Now, it would be a good idea to start the web application locally from our command-line terminal via the following command before we advance to the next subsection and deploy it on a public web server:

```
python3 app.py
```

After we have finished testing our application, we also shouldn't forget to remove the `debug=True` argument in the `app.run()` command of our `app.py` script (or set `debug=False`) as illustrated in the following figure:



```
            y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(~~debug=True~~)
```

Deploying the web application to a public server

After we have tested the web application locally, we are now ready to deploy our web application onto a public web server. For this tutorial, we will be using the PythonAnywhere web hosting service, which specializes in the hosting of Python web applications and makes it extremely simple and hassle-free. Furthermore, PythonAnywhere offers a beginner account option that lets us run a single web application free of charge.

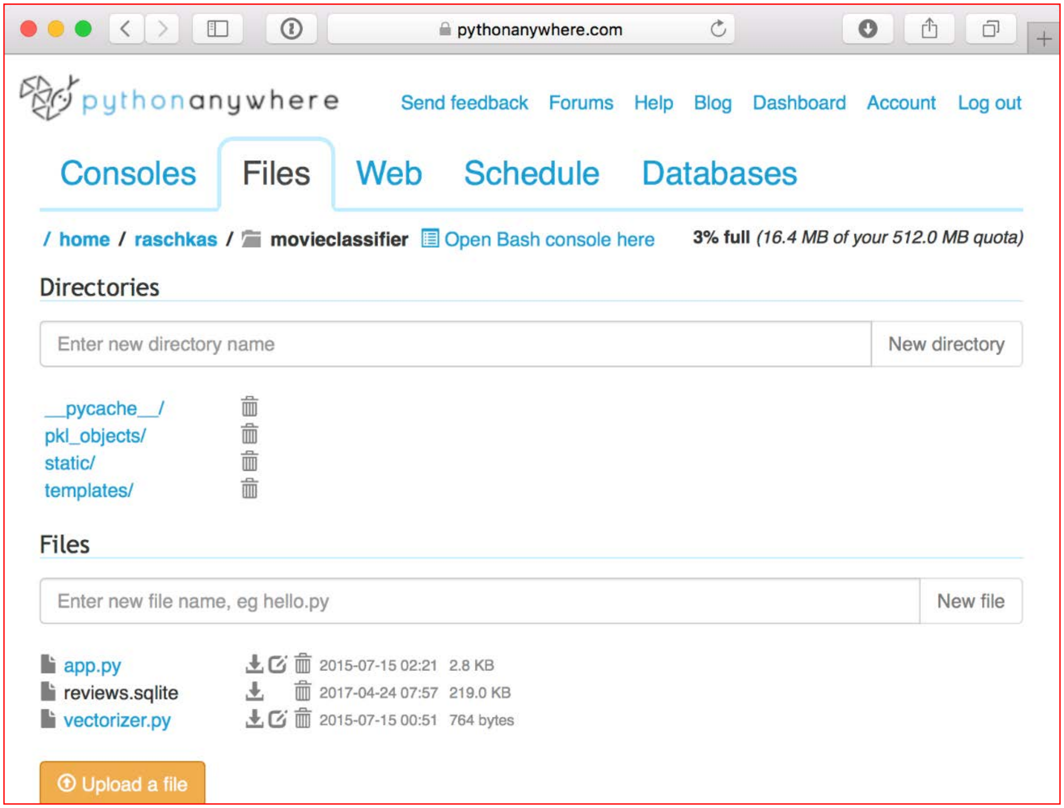
Creating a PythonAnywhere account

To create a new PythonAnywhere account, we visit the website at `https://www.pythonanywhere.com/` and click on the **Pricing & signup** link that is located in the top-right corner. Next, we click on the **Create a Beginner account** button where we need to provide a username, password, and valid email address. After we have read and agreed to the terms and conditions, we should have a new account.

Unfortunately, the free beginner account doesn't allow us to access the remote server via the Secure Socket Shell (SSH) protocol from our terminal. Thus, we need to use the PythonAnywhere web interface to manage our web application. But before we can upload our local application files to the server, we need to create a new web application for our PythonAnywhere account. After we click on the **Dashboard** button in the top-right corner, we have access to the control panel shown at the top of the page. Next, we click on the **Web** tab that is now visible at the top of the page. We proceed by clicking on the **+Add a new web app** button on the left, which lets us create a new Python 3.7 Flask web application that we name `movieclassifier`.

Uploading the movie classifier application

After creating a new application for our PythonAnywhere account, we head over to the **Files** tab to upload the files from our local `movieclassifier` directory using the PythonAnywhere web interface. After uploading the web application files that we created locally on our computer, we should have a `movieclassifier` directory in our PythonAnywhere account. It will contain the same directories and files as our local `movieclassifier` directory, as shown in the following screenshot:



/ home / raschkas / movieclassifier

3% full (16.4 MB of your 512.0 MB quota)

Directories

Enter new directory name

__pycache__/
pkl_objects/
static/
templates/

Files

Enter new file name, eg hello.py

```
app.py            2015-07-15 02:21  2.8 KB
reviews.sqlite    2017-04-24 07:57  219.0 KB
vectorizer.py     2015-07-15 00:51  764 bytes
```

Then, we head over to the **Web** tab one more time and click on the **Reload <username>.pythonanywhere.com** button to propagate the changes and refresh our web application. Finally, our web application should now be up and running and publicly available via `<username>.pythonanywhere.com`.

Troubleshooting

Unfortunately, web servers can be quite sensitive to the tiniest problems in our web application. If you are experiencing problems with running the web application on PythonAnywhere and are receiving error messages in your browser, you can check the server and error logs, which can be accessed from the **Web** tab in your PythonAnywhere account, to better diagnose the problem.

Updating the movie classifier

While our predictive model is updated on the fly whenever a user provides feedback about the classification, the updates to the `clf` object will be reset if the web server crashes or restarts. If we reload the web application, the `clf` object will be reinitialized from the `classifier.pkl` pickle file. One option to apply the updates permanently would be to pickle the `clf` object once again after each update. However, this would become computationally very inefficient with a growing number of users and could corrupt the pickle file if users provide feedback simultaneously.

An alternative solution is to update the predictive model from the feedback data that is being collected in the SQLite database. One option would be to download the SQLite database from the PythonAnywhere server, update the `clf` object locally on our computer, and upload the new pickle file to PythonAnywhere. To update the classifier locally on our computer, we create an `update.py` script file in the `movieclassifier` directory with the following contents:

```
import pickle
import sqlite3
import numpy as np
import os

# import HashingVectorizer from local dir
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):
    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
        model.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
    return model

cur_dir = os.path.dirname(__file__)

clf = pickle.load(open(os.path.join(cur_dir,
                  'pkl_objects',
                  'classifier.pkl'), 'rb'))

db = os.path.join(cur_dir, 'reviews.sqlite')

clf = update_model(db_path=db, model=clf, batch_size=10000)

# Uncomment the following lines if you are sure that
# you want to update your classifier.pkl file
# permanently.

# pickle.dump(clf, open(os.path.join(cur_dir,
#             'pkl_objects', 'classifier.pkl'), 'wb'),
#             protocol=4)
```

Getting the movieclassifier code files with the update functionality

A separate directory containing the movie review classifier application with the update functionality discussed in this chapter comes with the code examples for this book, which you can either obtain directly from Packt or download from GitHub at `https://github.com/rasbt/python-machine-learning-book-3rd-edition`. The code in this section is located in the `.../code/ch09/movieclassifier_with_update` subdirectory.

The `update_model` function will fetch entries from the SQLite database in batches of 10,000 entries at a time, unless the database contains fewer entries. Alternatively, we could also fetch one entry at a time by using `fetchone` instead of `fetchmany`, which would be computationally very inefficient. However, keep in mind that using the alternative `fetchall` method could be a problem if we are working with large datasets that exceed the computer or server's memory capacity.

Now that we have created the `update.py` script, we could also upload it to the `movieclassifier` directory on PythonAnywhere and import the `update_model` function in the main application script, `app.py`, to update the classifier from the SQLite database every time we restart the web application. In order to do so, we just need to add a line of code to import the `update_model` function from the `update.py` script at the top of `app.py`:

```
# import update function from local dir
from update import update_model
```

We then need to call the `update_model` function in the main application body:

```
...
if __name__ == '__main__':
    clf = update_model(db_path=db,
                       model=clf,
                       batch_size=10000)
...
```

As discussed, the modification in the previous code snippet will update the pickle file on PythonAnywhere. However, in practice, we do not often have to restart our web application, and it would make sense to validate the user feedback in the SQLite database prior to the update to make sure that the feedback is valuable information for the classifier.

Creating backups

In a real-world application, you may also want to back up the `classifier.pkl` pickle file periodically to have a safeguard against file corruption, for instance, by creating a timestamped version prior to each update. In order to create backups of the pickled classifier, you can import the following:

```
from shutil import copyfile
import time
```

Then, above the code that updates the pickled classifier, which is as follows,

```
pickle.dump(
    clf, open(
        os.path.join(
            cur_dir, 'pkl_objects',
            'classifier.pkl'),
        'wb'),
    protocol=4)
```

insert the following lines of code:

```
timestr = time.strftime("%Y%m%d-%H%M%S")
orig_path = os.path.join(
    cur_dir, 'pkl_objects', 'classifier.pkl')
backup_path = os.path.join(
    cur_dir, 'pkl_objects',
    'classifier_%s.pkl' % timestr)
copyfile(orig_path, backup_path)
```

As a consequence, backup files of the pickled classifier will be created following the format `YearMonthDay-HourMinuteSecond`, for example, `classifier_20190822-092148.pkl`.

Summary

In this chapter, you learned about many useful and practical topics that will extend your knowledge of machine learning theory. You learned how to serialize a model after training and how to load it for later use cases. Furthermore, we created an SQLite database for efficient data storage and created a web application that lets us make our movie classifier available to the outside world.

So far, in this book, we have covered many machine learning concepts, best practices, and supervised models for classification. In the next chapter, we will take a look at another subcategory of supervised learning, regression analysis, which lets us predict outcome variables on a continuous scale, in contrast to the categorical class labels of the classification models that we have been working with so far.

10

Predicting Continuous Target Variables with Regression Analysis

Throughout the previous chapters, you learned a lot about the main concepts behind **supervised learning** and trained many different models for classification tasks to predict group memberships or categorical variables. In this chapter, we will dive into another subcategory of supervised learning: **regression analysis**.

Regression models are used to predict target variables on a continuous scale, which makes them attractive for addressing many questions in science. They also have applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example is predicting the sales of a company in future months.

In this chapter, we will discuss the main concepts of regression models and cover the following topics:

- Exploring and visualizing datasets
- Looking at different approaches to implement linear regression models
- Training regression models that are robust to outliers
- Evaluating regression models and diagnosing common problems
- Fitting regression models to nonlinear data

Introducing linear regression

The goal of linear regression is to model the relationship between one or multiple features and a continuous target variable. In contrast to classification—a different subcategory of supervised learning—regression analysis aims to predict outputs on a continuous scale rather than categorical class labels.

In the following subsections, you will be introduced to the most basic type of linear regression, **simple linear regression**, and understand how to relate it to the more general, multivariate case (linear regression with multiple features).

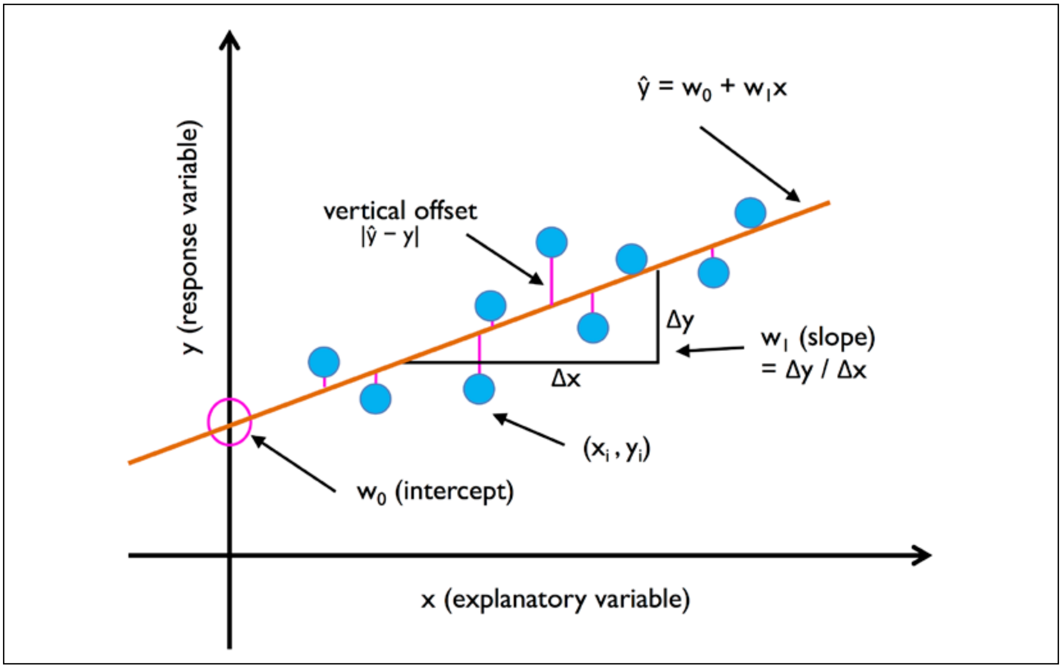
Simple linear regression

The goal of simple (**univariate**) linear regression is to model the relationship between a single feature (**explanatory variable**, *x*) and a continuous-valued **target** (**response variable**, *y*). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x$$

Here, the weight, w_0, represents the y axis intercept and w_1 is the weight coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the training examples, as shown in the following figure:



This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the training examples are the so-called **offsets** or **residuals**—the errors of our prediction.

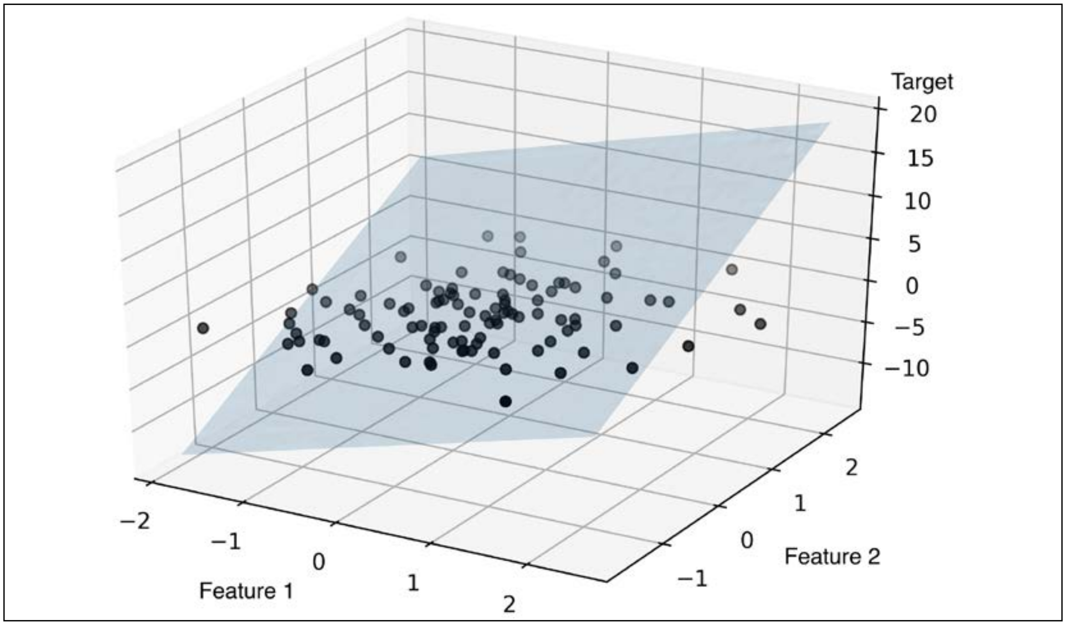
Multiple linear regression

The previous section introduced simple linear regression, a special case of linear regression with one explanatory variable. Of course, we can also generalize the linear regression model to multiple explanatory variables; this process is called **multiple linear regression**:

$$y = w_0x_0 + w_1x_1 + \ldots + w_mx_m = \sum_{i=0}^{n} w_ix_i = w^Tx$$

Here, w_0 is the y axis intercept with $x_0 = 1$.

The following figure shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look:



As you can see, visualizations of multiple linear regression hyperplanes in a three-dimensional scatterplot are already challenging to interpret when looking at static figures. Since we have no good means of visualizing hyperplanes with two dimensions in a scatterplot (multiple linear regression models fit to datasets with three or more features), the examples and visualizations in this chapter will mainly focus on the univariate case, using simple linear regression. However, simple and multiple linear regression are based on the same concepts and the same evaluation techniques; the code implementations that we will discuss in this chapter are also compatible with both types of regression model.

Exploring the Housing dataset

Before we implement the first linear regression model, we will discuss a new dataset, the Housing dataset, which contains information about houses in the suburbs of Boston collected by D. Harrison and D.L. Rubinfeld in 1978. The Housing dataset has been made freely available and is included in the code bundle of this book. The dataset has recently been removed from the UCI Machine Learning Repository but is available online at `https://raw.githubusercontent.com/rasbt/python-machine-learning-book-3rd-edition/master/ch10/housing.data.txt` or scikit-learn (`https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/datasets/data/boston_house_prices.csv`). As with each new dataset, it is always helpful to explore the data through a simple visualization, to get a better feeling of what we are working with.

Loading the Housing dataset into a data frame

In this section, we will load the Housing dataset using the pandas `read_csv` function, which is fast and versatile and a recommended tool for working with tabular data stored in a plaintext format.

The features of the 506 examples in the Housing dataset have been taken from the original source that was previously shared on `https://archive.ics.uci.edu/ml/datasets/Housing` and summarized here:

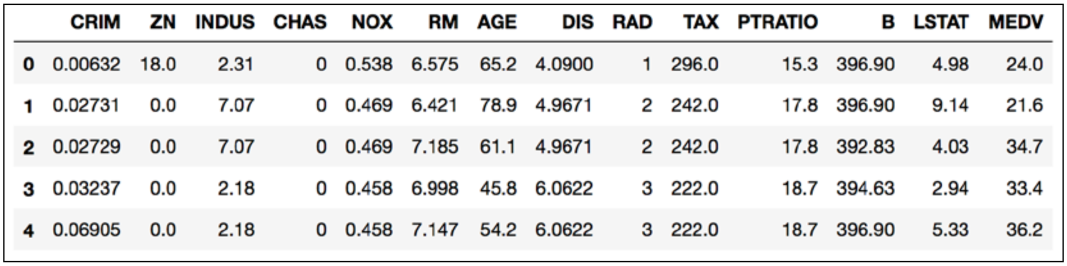
- `CRIM`: Per capita crime rate by town
- `ZN`: Proportion of residential land zoned for lots over 25,000 sq. ft.
- `INDUS`: Proportion of non-retail business acres per town
- `CHAS`: Charles River dummy variable (= 1 if tract bounds river and 0 otherwise)
- `NOX`: Nitric oxide concentration (parts per 10 million)

- RM: Average number of rooms per dwelling
- AGE: Proportion of owner-occupied units built prior to 1940
- DIS: Weighted distances to five Boston employment centers
- RAD: Index of accessibility to radial highways
- TAX: Full-value property tax rate per $10,000
- PTRATIO: Pupil-teacher ratio by town
- B: 1000(*Bk* – 0.63)^2, where *Bk* is the proportion of [people of African American descent] by town
- LSTAT: Percentage of lower status of the population
- MEDV: Median value of owner-occupied homes in $1000s

For the rest of this chapter, we will regard the house prices (MEDV) as our target variable—the variable that we want to predict using one or more of the 13 explanatory variables. Before we explore this dataset further, let's load it into a pandas DataFrame:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://raw.githubusercontent.com/rasbt/'
...                  'python-machine-learning-book-3rd-edition'
...                  '/master/ch10/housing.data.txt',
...                  header=None,
...                  sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

To confirm that the dataset was loaded successfully, we can display the first five lines of the dataset, as shown in the following figure:



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

Obtaining the Housing dataset

You can find a copy of the Housing dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the web link `https://raw.githubusercontent.com/rasbt/python-machine-learning-book-3rd-edition/master/code/ch10/housing.data.txt` is temporarily unavailable. For instance, to load the Housing dataset from a local directory, you can replace these lines:

```
df = pd.read_csv(
    'https://raw.githubusercontent.com/rasbt/'
    'python-machine-learning-book-3rd-edition/'
    'master/ch10/housing.data.txt',
    header=None,
    sep='\s+')
```

in the preceding code example with this line:

```
df = pd.read_csv('./housing.data.txt',
    sep='\s+')
```

Visualizing the important characteristics of a dataset

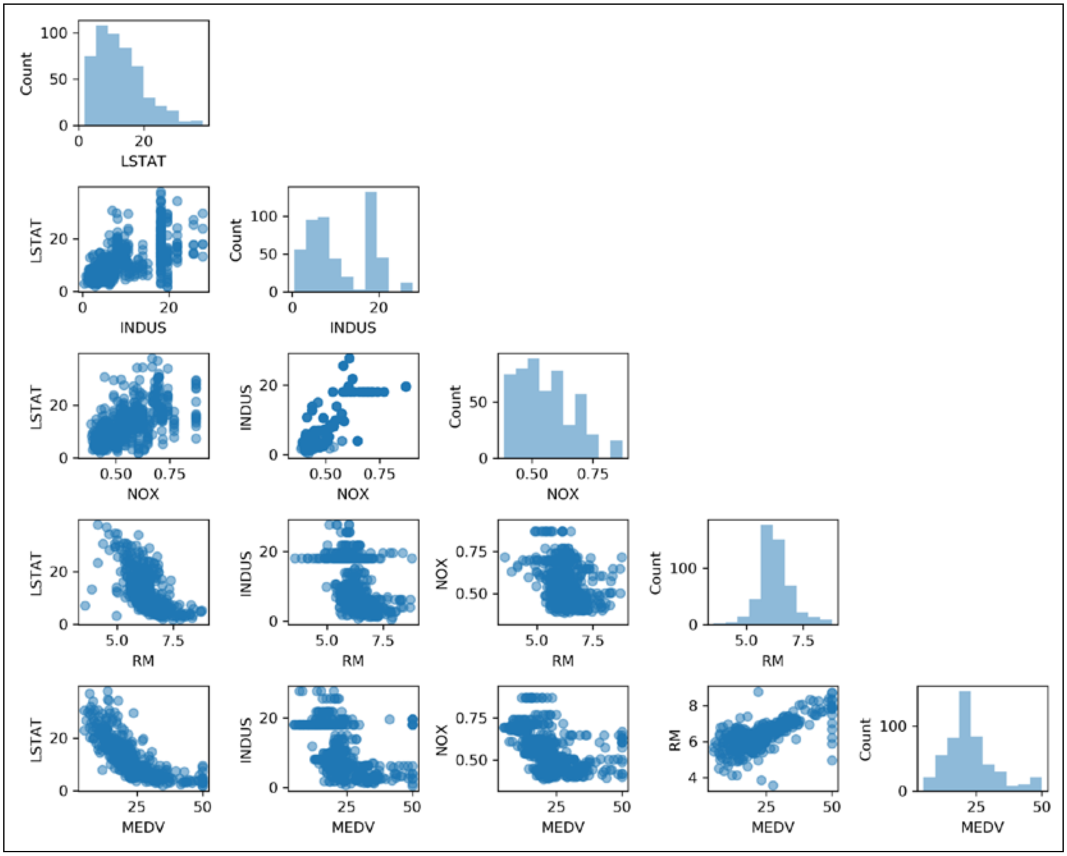
Exploratory data analysis (**EDA**) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, we will create a **scatterplot matrix** that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the `scatterplotmatrix` function from the MLxtend library (`http://rasbt.github.io/mlxtend/`), which is a Python library that contains various convenience functions for machine learning and data science applications in Python.

You can install the `mlxtend` package via `conda install mlxtend` or `pip install mlxtend`. After the installation is complete, you can import the package and create the scatterplot matrix as follows:

```
>>> import matplotlib.pyplot as plt
>>> from mlxtend.plotting import scatterplotmatrix
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> scatterplotmatrix(df[cols].values, figsize=(10, 8),
...                   names=cols, alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in the following figure, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:



Due to space constraints and in the interest of readability, we have only plotted five columns from the dataset: `LSTAT`, `INDUS`, `NOX`, `RM`, and `MEDV`. However, you are encouraged to create a scatterplot matrix of the whole `DataFrame` to explore the dataset further by choosing different column names in the previous `scatterplotmatrix` function call, or including all variables in the scatterplot matrix by omitting the column selector.

Using this scatterplot matrix, we can now quickly eyeball how the data is distributed and whether it contains outliers. For example, we can see that there is a linear relationship between `RM` and house prices, `MEDV` (the fifth column of the fourth row). Furthermore, we can see in the histogram—the lower-right subplot in the scatterplot matrix—that the `MEDV` variable seems to be normally distributed but contains several outliers.

Normality assumption of linear regression

Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistics and hypothesis tests that are beyond the scope of this book (for more information on this topic, please refer to *Introduction to Linear Regression Analysis*, *Montgomery*, *Douglas C. Montgomery*, *Elizabeth A. Peck*, and *G. Geoffrey Vining*, *Wiley*, *2012*, pages: 318-319).

Looking at relationships using a correlation matrix

In the previous section, we visualized the data distributions of the Housing dataset variables in the form of histograms and scatterplots. Next, we will create a correlation matrix to quantify and summarize linear relationships between variables. A correlation matrix is closely related to the covariance matrix that we covered in the section *Unsupervised dimensionality reduction via principal component analysis* in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. We can interpret the correlation matrix as being a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized features.

The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficient** (often abbreviated as **Pearson's r**), which measures the linear dependence between pairs of features. The correlation coefficients are in the range –1 to 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features, x and y (numerator), divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)}-\mu_x\right)\left(y^{(i)}-\mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)}-\mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)}-\mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

Here, μ denotes the mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations.

Covariance versus correlation for standardized features

We can show that the covariance between a pair of standardized features is, in fact, equal to their linear correlation coefficient. To show this, let's first standardize the features x and y to obtain their z-scores, which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we compute the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean zero, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x'^{(i)} - 0\right)\left(y'^{(i)} - 0\right)$$

Through resubstitution, we then get the following result:

$$\sigma'_{xy} \quad = \quad \frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\sigma'_{xy} \quad = \quad \frac{1}{n \cdot \sigma_x\sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

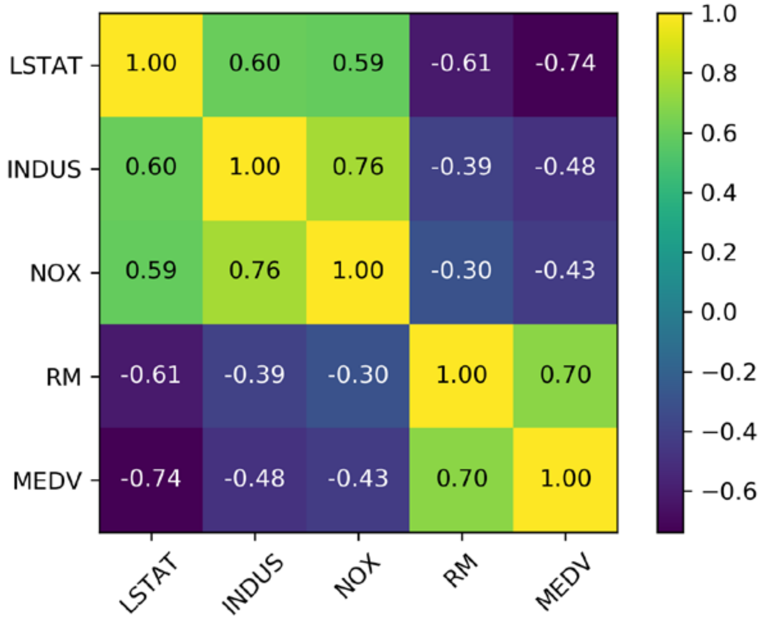
Finally, we can simplify this equation as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use MLxtend's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> from mlxtend.plotting import heatmap
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> hm = heatmap(cm,
...               row_names=cols,
...               column_names=cols)
>>> plt.show()
```

As you can see in the resulting figure, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:



To fit a linear regression model, we are interested in those features that have a high correlation with our target variable, MEDV. Looking at the previous correlation matrix, we can see that our target variable, MEDV, shows the largest correlation with the LSTAT variable (-0.74); however, as you might remember from inspecting the scatterplot matrix, there is a clear nonlinear relationship between LSTAT and MEDV. On the other hand, the correlation between RM and MEDV is also relatively high (0.70). Given the linear relationship between these two variables that we observed in the scatterplot, RM seems to be a good choice for an exploratory variable to introduce the concepts of a simple linear regression model in the following section.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, it was mentioned that linear regression can be understood as obtaining the best-fitting straight line through the examples of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **ordinary least squares** (**OLS**) method (sometimes also called **linear least squares**) to estimate the parameters of the linear regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the training examples.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **Adaptive Linear Neuron** (**Adaline**) from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. You will remember that the artificial neuron uses a linear activation function. Also, we defined a cost function, $J(\boldsymbol{w})$, which we minimized to learn the weights via optimization algorithms, such as **gradient descent** (**GD**) and **stochastic gradient descent** (**SGD**). This cost function in Adaline is the **sum of squared errors** (**SSE**), which is identical to the cost function that we use for OLS:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = \boldsymbol{w}^T \boldsymbol{x}$ (note that the term $\frac{1}{2}$ is just used for convenience to derive the update rule of GD). Essentially, OLS regression can be understood as Adaline without the unit step function so that we obtain continuous target values instead of the class labels `-1` and `1`. To demonstrate this, let's take the GD implementation of Adaline from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, and remove the unit step function to implement our first linear regression model:

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return self.net_input(X)
```

Weight updates with gradient descent

If you need a refresher about how the weights are updated—taking a step in the opposite direction of the gradient—please revisit the *Adaptive linear neurons and the convergence of learning* section in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*.

To see our `LinearRegressionGD` regressor in action, let's use the `RM` (number of rooms) variable from the Housing dataset as the explanatory variable and train a model that can predict `MEDV` (house prices). Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

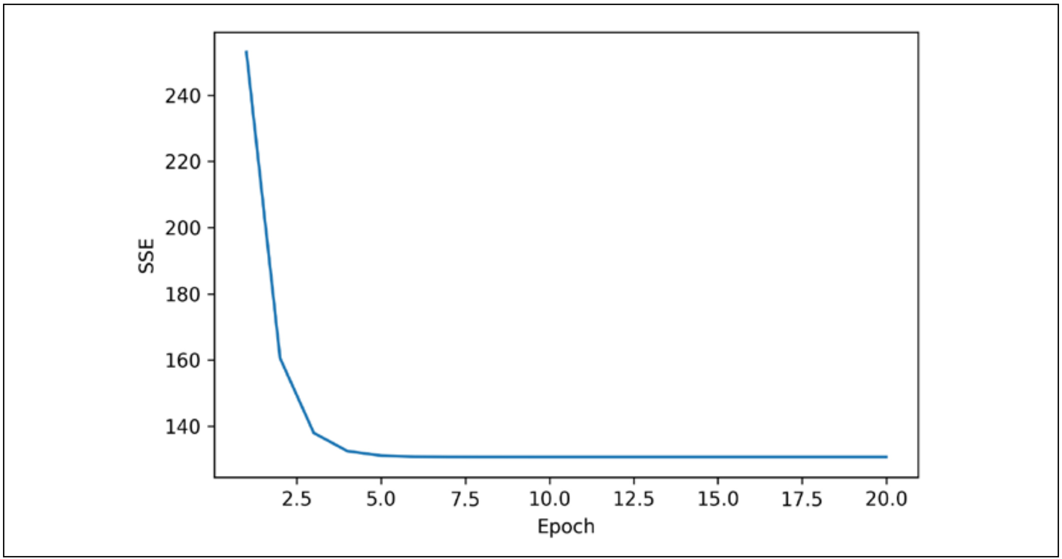
```
>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)
```

Notice the workaround regarding `y_std`, using `np.newaxis` and `flatten`. Most transformers in scikit-learn expect data to be stored in two-dimensional arrays. In the previous code example, the use of `np.newaxis` in `y[:, np.newaxis]` added a new dimension to the array. Then, after `StandardScaler` returned the scaled variable, we converted it back to the original one-dimensional array representation using the `flatten()` method for our convenience.

We discussed in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, that it is always a good idea to plot the cost as a function of the number of epochs (complete iterations) over the training dataset when we are using optimization algorithms, such as GD, to check that the algorithm converged to a cost minimum (here, a *global* cost minimum):

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As you can see in the following plot, the GD algorithm converged after the fifth epoch:



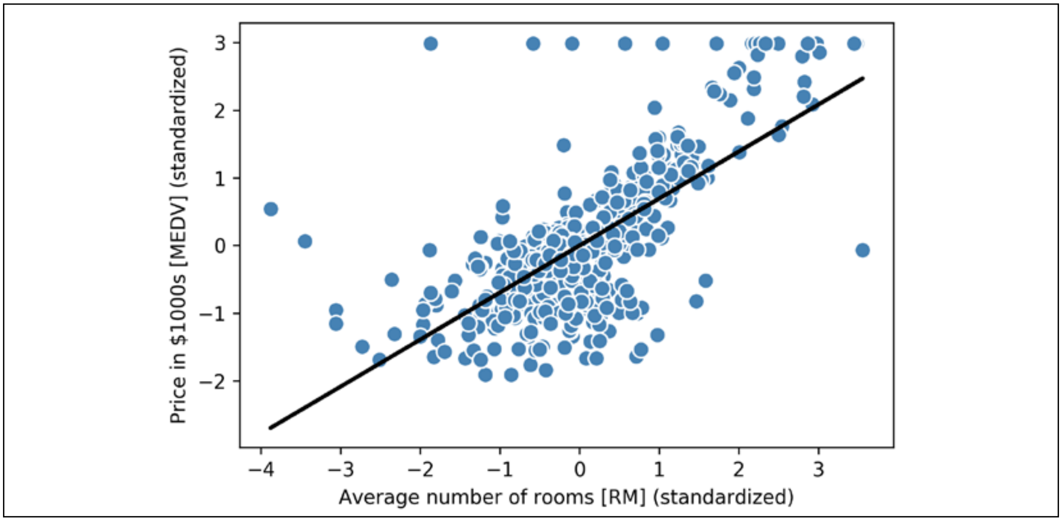
Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training examples and add the regression line:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
...     return None
```

Now, we will use this `lin_regplot` function to plot the number of rooms against the house price:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Average number of rooms [RM] (standardized)')
>>> plt.ylabel('Price in $1000s [MEDV] (standardized)')
>>> plt.show()
```

As you can see in the following plot, the linear regression line reflects the general trend that house prices tend to increase with the number of rooms:



Although this observation makes sense, the data also tells us that the number of rooms does not explain house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we can also observe that several data points lined up at $y = 3$, which suggests that the prices may have been clipped. In certain applications, it may also be important to report the predicted outcome variables on their original scale. To scale the predicted price outcome back onto the `Price in $1000s` axis, we can simply apply the `inverse_transform` method of the `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform(np.array([[5.0]]))
>>> price_std = lr.predict(num_rooms_std)
>>> print("Price in $1000s: %.3f" % \
...        sc_y.inverse_transform(price_std))
Price in $1000s: 10.840
```

In this code example, we used the previously trained linear regression model to predict the price of a house with five rooms. According to our model, such a house will be worth $10,840.

As a side note, it is also worth mentioning that we technically don't have to update the weights of the intercept if we are working with standardized variables, since the y axis intercept is always 0 in those cases. We can quickly confirm this by printing the weights:

```
>>> print('Slope: %.3f' % lr.w_[1])
Slope: 0.695
>>> print('Intercept: %.3f' % lr.w_[0])
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

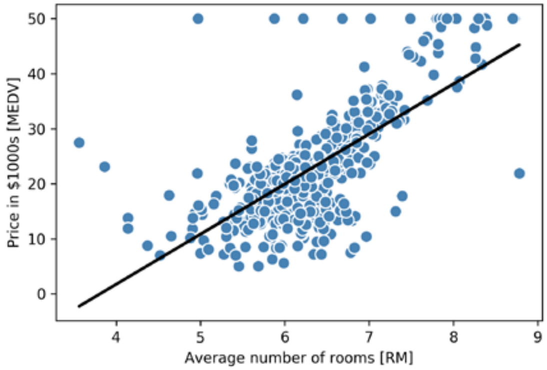
In the previous section, we implemented a working model for regression analysis; however, in a real-world application, we may be interested in more efficient implementations. For example, many of scikit-learn's estimators for regression make use of the least squares implementation in SciPy (`scipy.linalg.lstsq`), which in turn uses highly optimized code optimizations based on the Linear Algebra Package (LAPACK). The linear regression implementation in scikit-learn also works (better) with unstandardized variables, since it does not use (S)GD-based optimization, so we can skip the standardization step:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> y_pred = slr.predict(X)
>>> print('Slope: %.3f' % slr.coef_[0])
Slope: 9.102
>>> print('Intercept: %.3f' % slr.intercept_)
Intercept: -34.671
```

As you can see from executing this code, scikit-learn's `LinearRegression` model, fitted with the unstandardized `RM` and `MEDV` variables, yielded different model coefficients, since the features have not been standardized. However, when we compare it to our GD implementation by plotting `MEDV` against `RM`, we can qualitatively see that it fits the data similarly well:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000s [MEDV]')
>>> plt.show()
```

For instance, we can see that the overall result looks identical to our GD implementation:



Analytical solutions of linear regression

As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Slope: %.3f' % w[1])
Slope: 9.102
>>> print('Intercept: %.3f' % w[0])
Intercept: -34.671
```

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the **normal equation**), or the matrix containing the training examples may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain normal equations, take a look at Dr. Stephen Pollock's chapter *The Classical Linear Regression Model* from his lectures at the University of Leicester, which is available for free at: `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

Also, if you want to compare linear regression solutions obtained via GD, SGD, the closed-form solution, QR factorization, and singular vector decomposition, you can use the `LinearRegression` class implemented in MLxtend (`http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/`), which lets users toggle between these options. Another great library to recommend for regression modeling in Python is Statsmodels, which implements more advanced linear regression models, as illustrated at `https://www.statsmodels.org/stable/examples/index.html#regression`.

Fitting a robust regression model using RANSAC

Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. There are many statistical tests that can be used to detect outliers, which are beyond the scope of the book. However, removing outliers always requires our own judgment as data scientists as well as our domain knowledge.

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called **inliers**.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of examples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations were reached; go back to step 1 otherwise.

Let's now use a linear model in combination with the RANSAC algorithm as implemented in scikit-learn's `RANSACRegressor` class:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                          max_trials=100,
...                          min_samples=50,
...                          loss='absolute_loss',
...                          residual_threshold=5.0,
...                          random_state=0)
>>> ransac.fit(X, y)
```

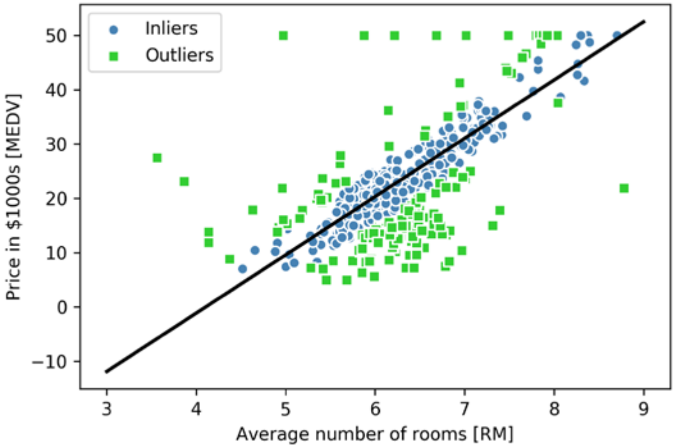
We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=50`, we set the minimum number of the randomly chosen training examples to be at least 50. Using `'absolute_loss'` as an argument for the `loss` parameter, the algorithm computes absolute vertical distances between the fitted line and the training examples. By setting the `residual_threshold` parameter to `5.0`, we only allow training examples to be included in the inlier set if their vertical distance to the fitted line is within 5 distance units, which works well on this particular dataset.

By default, scikit-learn uses the **MAD** estimate to select the inlier threshold, where MAD stands for the **median absolute deviation** of the target values, `y`. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC. Many different approaches have been developed in recent years to select a good inlier threshold automatically. You can find a detailed discussion in *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting, R. Toldo, A. Fusiello's, Springer, 2009* (in *Image Analysis and Processing–ICIAP 2009*, pages: 123-131).

Once we have fitted the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC-linear regression model and plot them together with the linear fit:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='steelblue', edgecolor='white',
...             marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='limegreen', edgecolor='white',
...             marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
>>> plt.xlabel('Average number of rooms [RM]')
>>> plt.ylabel('Price in $1000s [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

As you can see in the following scatterplot, the linear regression model was fitted on the detected set of inliers, which are shown as circles:



When we print the slope and intercept of the model by executing the following code, the linear regression line will be slightly different from the fit that we obtained in the previous section without using RANSAC:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 10.735
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -44.089
```

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know whether this approach will have a positive effect on the predictive performance for unseen data or not. Thus, in the next section, we will look at different approaches for evaluating a regression model, which is a crucial part of building systems for predictive modeling.

Evaluating the performance of linear regression models

In the previous section, you learned how to fit a regression model on training data. However, you discovered in previous chapters that it is crucial to test the model on data that it hasn't seen during training to obtain a more unbiased estimate of its generalization performance.

As you will remember from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we want to split our dataset into separate training and test datasets, where we will use the former to fit the model and the latter to evaluate its performance on unseen data to estimate the generalization performance. Instead of proceeding with the simple regression model, we will now use all variables in the dataset and train a multiple regression model:

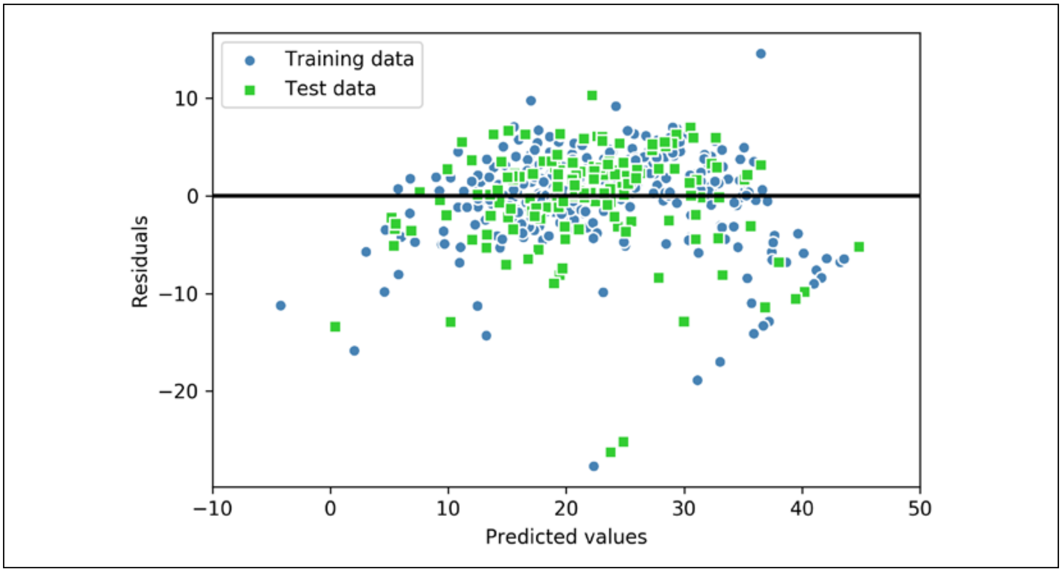
```
>>> from sklearn.model_selection import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...       X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Since our model uses multiple explanatory variables, we can't visualize the linear regression line (or hyperplane, to be precise) in a two-dimensional plot, but we can plot the residuals (the differences or vertical distances between the actual and predicted values) versus the predicted values to diagnose our regression model. **Residual plots** are a commonly used graphical tool for diagnosing regression models. They can help to detect nonlinearity and outliers, and check whether the errors are randomly distributed.

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...             c='steelblue', marker='o', edgecolor='white',
...             label='Training data')
>>> plt.scatter(y_test_pred, y_test_pred - y_test,
...             c='limegreen', marker='s', edgecolor='white',
...             label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, color='black', lw=2)
>>> plt.xlim([-10, 50])
>>> plt.show()
```

After executing the code, we should see a residual plot with a line passing through the *x* axis origin, as shown here:



In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect the errors to be randomly distributed and the residuals to be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which has leaked into the residuals, as you can slightly see in our previous residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the so-called **mean squared error** (**MSE**), which is simply the averaged value of the SSE cost that we minimized to fit the linear regression model. The MSE is useful for comparing different regression models or for tuning their parameters via grid search and cross-validation, as it normalizes the SSE by the sample size:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Let's compute the MSE of our training and test predictions:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE train: %.3f, test: %.3f' % (
...        mean_squared_error(y_train, y_train_pred),
...        mean_squared_error(y_test, y_test_pred)))
MSE train: 19.958, test: 27.196
```

You can see that the MSE on the training dataset is 19.96, and the MSE on the test dataset is much larger, with a value of 27.20, which is an indicator that our model is overfitting the training data in this case. However, please be aware that the MSE is unbounded in contrast to the classification accuracy, for example. In other words, the interpretation of the MSE depends on the dataset and feature scaling. For example, if the house prices were presented as multiples of 1,000 (with the K suffix), the same model would yield a lower MSE compared to a model that worked with unscaled features. To further illustrate this point, $(\$10\text{K} - 15\text{K})^2 < (\$10{,}000 - \$15{,}000)^2$.

Thus, it may sometimes be more useful to report the **coefficient of determination** (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or, in other words, R^2 is the fraction of response variance that is captured by the model. The R^2 value is defined as:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors and SST is the total sum of squares:

$$SST = \sum_{i=1}^{n} \left(y^{(i)} - \mu_y \right)^2$$

In other words, SST is simply the variance of the response.

Let's quickly show that R^2 is indeed just a rescaled version of the MSE:

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} \\ &= \frac{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

For the training dataset, the R^2 is bounded between 0 and 1, but it can become negative for the test dataset. If $R^2 = 1$, the model fits the data perfectly with a corresponding *MSE* = 0.

Evaluated on the training data, the R^2 of our model is 0.765, which doesn't sound too bad. However, the R^2 on the test dataset is only 0.673, which we can compute by executing the following code:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 train: %.3f, test: %.3f' %
...       (r2_score(y_train, y_train_pred),
...        r2_score(y_test, y_test_pred)))
R^2 train: 0.765, test: 0.673
```

Using regularized methods for regression

As we discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, regularization is one approach to tackling the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called **Ridge Regression**, **least absolute shrinkage and selection operator** (**LASSO**), and **elastic Net**.

Ridge Regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:

$$J(\boldsymbol{w})_{Ridge} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|\boldsymbol{w}\|_2^2$$

Here:

$$L2\colon \lambda\|\boldsymbol{w}\|_2^2 = \lambda\sum_{j=1}^{m} w_j{}^2$$

By increasing the value of hyperparameter λ, we increase the regularization strength and thereby shrink the weights of our model. Please note that we don't regularize the intercept term, w_0.

An alternative approach that can lead to sparse models is LASSO. Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique:

$$J(\boldsymbol{w})_{LASSO} = \sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda\|\boldsymbol{w}\|_1$$

Here, the L1 penalty for LASSO is defined as the sum of the absolute magnitudes of the model weights, as follows:

$$L1\colon \lambda\|\boldsymbol{w}\|_1 = \lambda\sum_{j=1}^{m}\left|w_j\right|$$

However, a limitation of LASSO is that it selects at most *n* features if *m* > *n*, where *n* is the number of training examples. This may be undesirable in certain applications of feature selection. In practice, however, this property of LASSO is often an advantage because it avoids saturated models. Saturation of a model occurs if the number of training examples is equal to the number of features, which is a form of overparameterization. As a consequence, a saturated model can always fit the training data perfectly but is merely a form of interpolation and thus is not expected to generalize well.

A compromise between Ridge Regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than *n* features if *m* > *n*:

$$J(\boldsymbol{w})_{ElasticNet} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} {w_j}^2 + \lambda_2 \sum_{j=1}^{m} \left|w_j\right|$$

Those regularized regression models are all available via scikit-learn, and their usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

A Ridge Regression model can be initialized via:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter λ. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set the `l1_ratio` to 1.0, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please see the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d$$

Here, *d* denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients, ***w***. In the following subsections, we will see how we can add such polynomial terms to an existing dataset conveniently and fit a polynomial regression model.

Adding polynomial terms using scikit-learn

We will now learn how to use the `PolynomialFeatures` transformer class from scikit-learn to add a quadratic term (*d* = 2) to a simple regression problem with one explanatory variable. Then, we will compare the polynomial to the linear fit by following these steps:

1. Add a second-degree polynomial term:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...                368.0, 396.0, 446.0, 480.0, 586.0])\
...               [:, np.newaxis]
>>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...                342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Fit a simple linear regression model for comparison:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

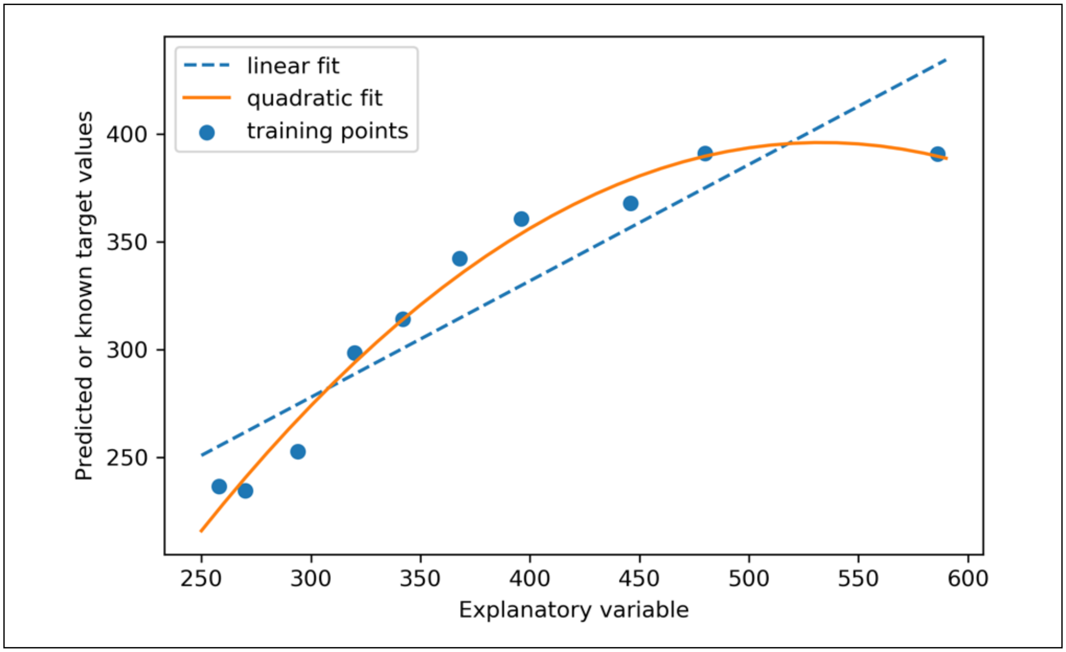
3. Fit a multiple regression model on the transformed features for polynomial regression:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Plot the results:

```
>>> plt.scatter(X, y, label='Training points')
>>> plt.plot(X_fit, y_lin_fit,
...          label='Linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...          label='Quadratic fit')
>>> plt.xlabel('Explanatory variable')
>>> plt.ylabel('Predicted or known target values')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In the resulting plot, you can see that the polynomial fit captures the relationship between the response and explanatory variables much better than the linear fit:



Next, we will compute the MSE and R^2 evaluation metrics:

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('Training MSE linear: %.3f, quadratic: %.3f' % (
...       mean_squared_error(y, y_lin_pred),
...       mean_squared_error(y, y_quad_pred)))
Training MSE linear: 569.780, quadratic: 61.330
>>> print('Training R^2 linear: %.3f, quadratic: %.3f' % (
...       r2_score(y, y_lin_pred),
...       r2_score(y, y_quad_pred)))
Training R^2 linear: 0.832, quadratic: 0.982
```

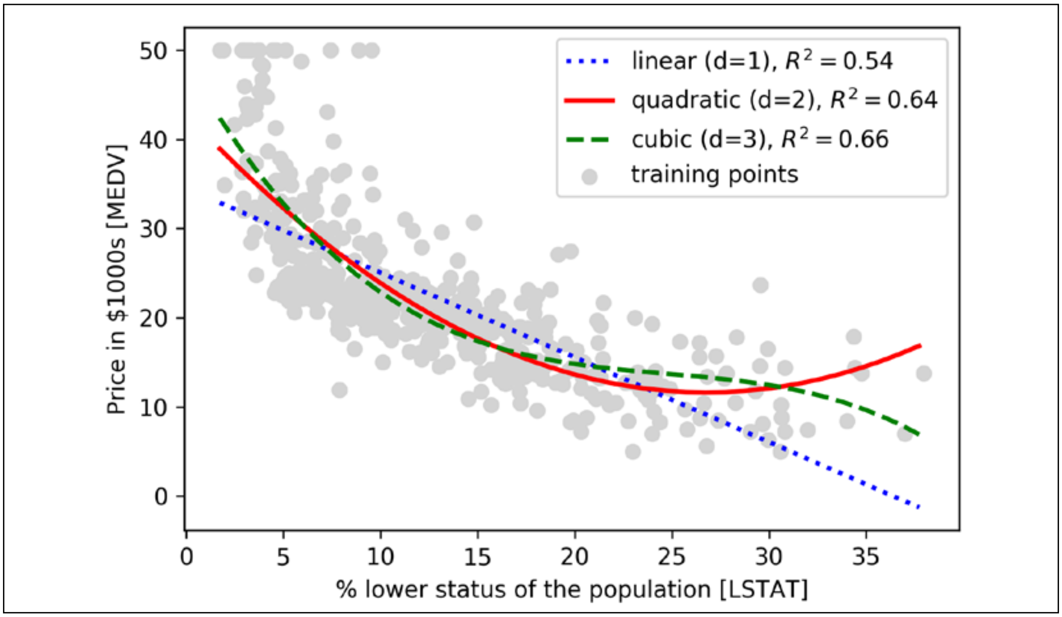
As you can see after executing the code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit); also, the coefficient of determination reflects a closer fit of the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Housing dataset

In the preceding subsection, you learned how to construct polynomial features to fit nonlinear relationships in a toy problem; let's now take a look at a more concrete example and apply those concepts to the data in the Housing dataset. By executing the following code, we will model the relationship between house prices and `LSTAT` (percentage of lower status of the population) using second-degree (quadratic) and third-degree (cubic) polynomials and compare that to a linear fit:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values

>>> regr = LinearRegression()

# create quadratic features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

# fit features
>>> X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]

>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))

>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))

>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

# plot results
>>> plt.scatter(X, y, label='Training points', color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='Linear (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')

>>> plt.plot(X_fit, y_quad_fit,
...          label='Quadratic (d=2), $R^2=%.2f$' % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')

>>> plt.plot(X_fit, y_cubic_fit,
...          label='Cubic (d=3), $R^2=%.2f$' % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')

>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000s [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()
```

The resulting plot is as follows:



As you can see, the cubic fit captures the relationship between house prices and LSTAT better than the linear and quadratic fit. However, you should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.

In addition, polynomial features are not always the best choice for modeling nonlinear relationships. For example, with some experience or intuition, just looking at the MEDV-LSTAT scatterplot may lead to the hypothesis that a log-transformation of the LSTAT feature variable and the square root of MEDV may project the data onto a linear feature space suitable for a linear regression fit. For instance, my perception is that this relationship between the two variables looks quite similar to an exponential function:

$$f(x) = e^{-x}$$

Since the natural logarithm of an exponential function is a straight line, I assume that such a log-transformation can be usefully applied here:

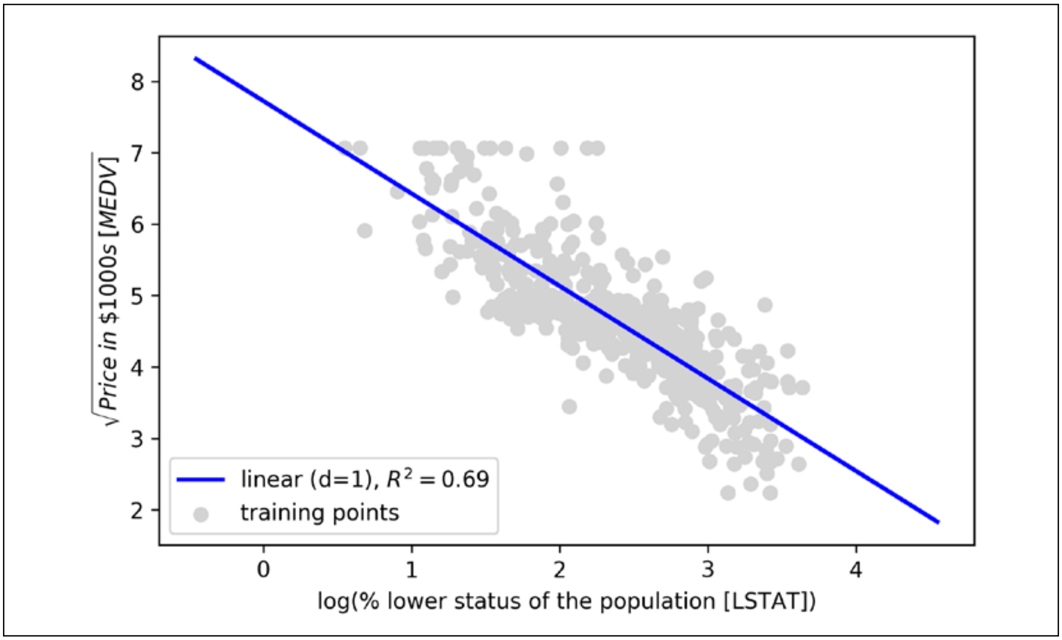
$$\log\big(f(x)\big) = -x$$

Let's test this hypothesis by executing the following code:

```
>>> # transform features
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)
>>>
>>> # fit features
>>> X_fit = np.arange(X_log.min()-1,
...                    X_log.max()+1, 1)[:, np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

>>> # plot results
>>> plt.scatter(X_log, y_sqrt,
...             label='Training points',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='Linear (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2)
```

```
>>> plt.xlabel('log(% lower status of the population [LSTAT])')
>>> plt.ylabel('$\sqrt{Price \; in \; \$1000s \; [MEDV]}$')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

After transforming the explanatory onto the log space and taking the square root of the target variables, we were able to capture the relationship between the two variables with a linear regression line that seems to fit the data better ($R^2 = 0.69$) than any of the previous polynomial feature transformations:



Dealing with nonlinear relationships using random forests

In this section, we are going to take a look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions, in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we subdivide the input space into smaller regions that become more manageable.

Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data, because decision trees analyze one feature at a time, rather than taking weighted combinations into account. (Likewise, normalizing or standardizing features is not required for decision trees.) You will remember from *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, that we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **information gain** (**IG**), which can be defined as follows for a binary split:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Here, x_i is the feature to perform the split, N_p is the number of training examples in the parent node, *I* is the impurity function, D_p is the subset of training examples at the parent node, and D_{left} and D_{right} are the subsets of training examples at the left and right child nodes after the split. Remember that our goal is to find the feature split that maximizes the information gain; in other words, we want to find the feature split that reduces the impurities in the child nodes most. In *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we discussed Gini impurity and entropy as measures of impurity, which are both useful criteria for classification. To use a decision tree for regression, however, we need an impurity metric that is suitable for continuous variables, so we define the impurity measure of a node, *t*, as the MSE instead:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t\right)^2$$

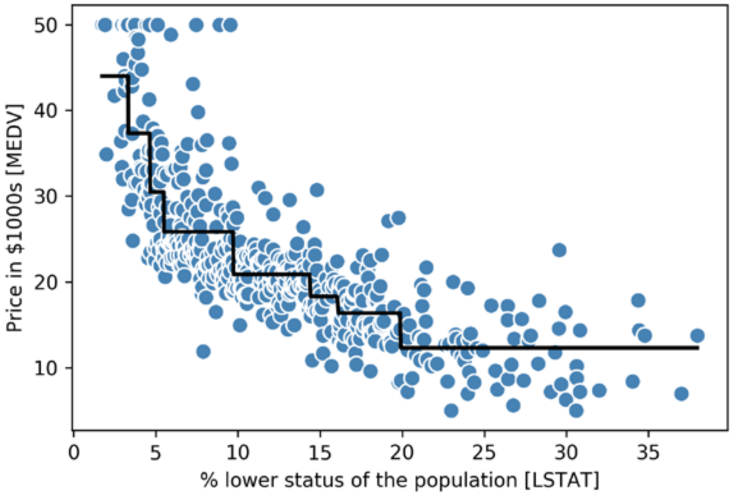
Here, N_t is the number of training examples at node *t*, D_t is the training subset at node *t*, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often referred to as **within-node variance**, which is why the splitting criterion is also better known as **variance reduction**. To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the nonlinear relationship between the `MEDV` and `LSTAT` variables:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% lower status of the population [LSTAT]')
>>> plt.ylabel('Price in $1000s [MEDV]')
>>> plt.show()
```

As you can see in the resulting plot, the decision tree captures the general trend in the data. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree so as to not overfit or underfit the data; here, a depth of three seemed to be a good choice:



In the next section, we will take a look at a more robust way of fitting regression trees: random forests.

Random forest regression

As you learned in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, the random forest algorithm is an ensemble technique that combines multiple decision trees. A random forest usually has a better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble. The basic random forest algorithm for regression is almost identical to the random forest algorithm for classification that we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*. The only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction over all decision trees.

Now, let's use all the features in the Housing dataset to fit a random forest regression model on 60 percent of the examples and evaluate its performance on the remaining 40 percent. The code is as follows:

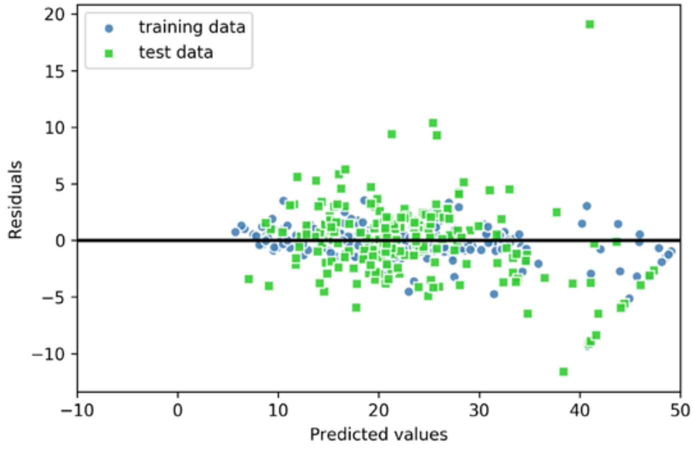
```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.4,
...                      random_state=1)
>>>
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(n_estimators=1000,
...                                criterion='mse',
...                                random_state=1,
...                                n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE train: %.3f, test: %.3f' % (
...       mean_squared_error(y_train, y_train_pred),
...       mean_squared_error(y_test, y_test_pred)))
MSE train: 1.642, test: 11.052
>>> print('R^2 train: %.3f, test: %.3f' % (
...       r2_score(y_train, y_train_pred),
...       r2_score(y_test, y_test_pred)))
R^2 train: 0.979, test: 0.878
```

Unfortunately, you can see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.878$ on the test dataset).

Lastly, let's also take a look at the residuals of the prediction:

```
>>> plt.scatter(y_train_pred,
...             y_train_pred - y_train,
...             c='steelblue',
...             edgecolor='white',
...             marker='o',
...             s=35,
...             alpha=0.9,
...             label='Training data')
>>> plt.scatter(y_test_pred,
...             y_test_pred - y_test,
...             c='limegreen',
...             edgecolor='white',
...             marker='s',
...             s=35,
...             alpha=0.9,
...             label='Test data')
>>> plt.xlabel('Predicted values')
>>> plt.ylabel('Residuals')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
>>> plt.xlim([-10, 50])
>>> plt.tight_layout()
>>> plt.show()
```

As it was already summarized by the R^2 coefficient, you can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter:



Ideally, our model error should be random or unpredictable. In other words, the error of the predictions should not be related to any of the information contained in the explanatory variables; rather, it should reflect the randomness of the real-world distributions or patterns. If we find patterns in the prediction errors, for example, by inspecting the residual plot, it means that the residual plots contain predictive information. A common reason for this could be that explanatory information is leaking into those residuals.

Unfortunately, there is not a universal approach for dealing with non-randomness in residual plots, and it requires experimentation. Depending on the data that is available to us, we may be able to improve the model by transforming variables, tuning the hyperparameters of the learning algorithm, choosing simpler or more complex models, removing outliers, or including additional variables.

Regression with support vector machines

In *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*, we also learned about the kernel trick, which can be used in combination with a **support vector machine** (**SVM**) for classification, and is useful if we are dealing with nonlinear problems. Although a discussion is beyond the scope of this book, SVMs can also be used in nonlinear regression tasks. The interested reader can find more information about SVMs for regression in an excellent report: *Support Vector Machines for Classification and Regression*, *S. R. Gunn* and others, *University of Southampton technical report*, 14, *1998* (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.579.6867&rep=rep1&type=pdf`). An SVM regressor is also implemented in scikit-learn, and more information about its usage can be found at `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR`.

Summary

At the beginning of this chapter, you learned about simple linear regression analysis to model the relationship between a single explanatory variable and a continuous response variable. We then discussed a useful explanatory data analysis technique to look at patterns and anomalies in data, which is an important first step in predictive modeling tasks.

We built our first model by implementing linear regression using a gradient-based optimization approach. You then saw how to utilize scikit-learn's linear models for regression and also implement a robust regression technique (RANSAC) as an approach for dealing with outliers. To assess the predictive performance of regression models, we computed the mean sum of squared errors and the related R^2 metric. Furthermore, we also discussed a useful graphical approach to diagnose the problems of regression models: the residual plot.

After we explored how regularization can be applied to regression models to reduce the model complexity and avoid overfitting, we also covered several approaches to model nonlinear relationships, including polynomial feature transformation and random forest regressors.

We discussed supervised learning, classification, and regression analysis in great detail in the previous chapters. In the next chapter, we are going to learn about another interesting subfield of machine learning, unsupervised learning, and also how to use cluster analysis to find hidden structures in data in the absence of target variables.

11
Working with Unlabeled Data – Clustering Analysis

In the previous chapters, we used supervised learning techniques to build machine learning models, using data where the answer was already known—the class labels were already available in our training data. In this chapter, we will switch gears and explore cluster analysis, a category of **unsupervised learning** techniques that allows us to discover hidden structures in data where we do not know the right answer upfront. The goal of **clustering** is to find a natural grouping in data so that items in the same cluster are more similar to each other than to those from different clusters.

Given its exploratory nature, clustering is an exciting topic, and in this chapter, you will learn about the following concepts, which can help us to organize data into meaningful structures:

- Finding centers of similarity using the popular **k-means** algorithm
- Taking a bottom-up approach to building hierarchical clustering trees
- Identifying arbitrary shapes of objects using a density-based clustering approach

Grouping objects by similarity using k-means

In this section, we will learn about one of the most popular clustering algorithms, k-means, which is widely used in academia as well as in industry. Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines.

K-means clustering using scikit-learn

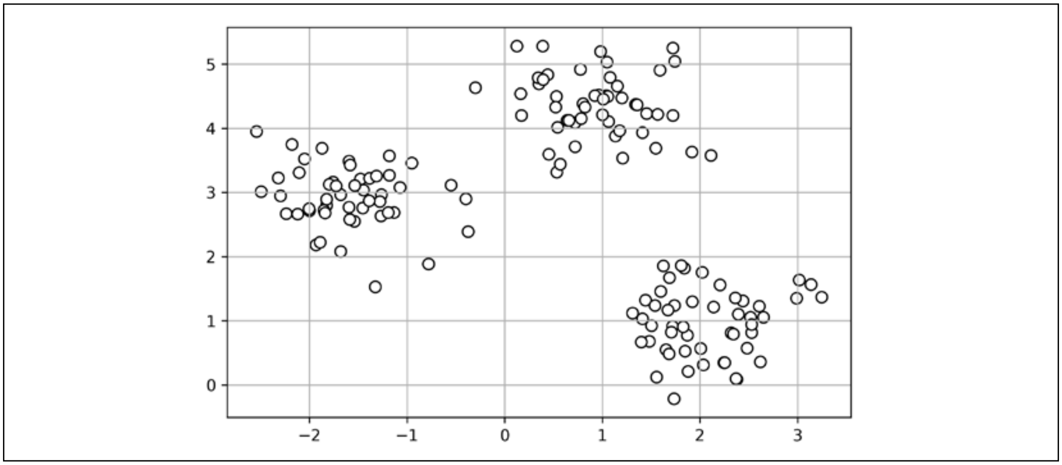
As you will see in a moment, the k-means algorithm is extremely easy to implement, but it is also computationally very efficient compared to other clustering algorithms, which might explain its popularity. The k-means algorithm belongs to the category of **prototype-based clustering**. We will discuss two other categories of clustering, **hierarchical** and **density-based clustering**, later in this chapter.

Prototype-based clustering means that each cluster is represented by a prototype, which is usually either the **centroid** (*average*) of similar points with continuous features, or the **medoid** (the most *representative* or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features. While k-means is very good at identifying clusters with a spherical shape, one of the drawbacks of this clustering algorithm is that we have to specify the number of clusters, *k*, *a priori*. An inappropriate choice for *k* can result in poor clustering performance. Later in this chapter, we will discuss the **elbow** method and **silhouette plots**, which are useful techniques to evaluate the quality of a clustering to help us determine the optimal number of clusters, *k*.

Although k-means clustering can be applied to data in higher dimensions, we will walk through the following examples using a simple two-dimensional dataset for the purpose of visualization:

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                   n_features=2,
...                   centers=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)

>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:, 0],
...             X[:, 1],
...             c='white',
...             marker='o',
...             edgecolor='black',
...             s=50)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot:



In real-world applications of clustering, we do not have any ground truth category information (information provided as empirical evidence as opposed to inference) about those examples; if we were given class labels, this task would fall into the category of supervised learning. Thus, our goal is to group the examples based on their feature similarities, which can be achieved using the k-means algorithm, as summarized by the following four steps:

1. Randomly pick *k* centroids from the examples as initial cluster centers.
2. Assign each example to the nearest centroid, $\mu^{(j)}, j \in \{1, \ldots, k\}$.
3. Move the centroids to the center of the examples that were assigned to it.
4. Repeat steps 2 and 3 until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached.

Now, the next question is, *how do we measure similarity between objects*? We can define similarity as the opposite of distance, and a commonly used distance for clustering examples with continuous features is the **squared Euclidean distance** between two points, ***x*** and ***y***, in *m*-dimensional space:

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} (x_j - y_j)^2 = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$$

Note that, in the preceding equation, the index *j* refers to the *j*th dimension (feature column) of the example inputs, ***x*** and ***y***. In the rest of this section, we will use the superscripts *i* and *j* to refer to the index of the example (data record) and cluster index, respectively.

Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the within-cluster **sum of squared errors** (**SSE**), which is sometimes also called **cluster inertia**:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

Here, $\boldsymbol{\mu}^{(j)}$ is the representative point (centroid) for cluster *j*. $w^{(i,j)} = 1$ if the example, $\boldsymbol{x}^{(i)}$, is in cluster *j*, or 0 otherwise.

$$w^{(i,j)} = \begin{cases} 1, & \text{if } x^{(i)} \in j \\ 0, & \text{otherwise} \end{cases}$$

Now that you have learned how the simple k-means algorithm works, let's apply it to our example dataset using the `KMeans` class from scikit-learn's `cluster` module:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

Using the preceding code, we set the number of desired clusters to `3`; specifying the number of clusters *a priori* is one of the limitations of k-means. We set `n_init=10` to run the k-means clustering algorithms 10 times independently, with different random centroids to choose the final model as the one with the lowest SSE. Via the `max_iter` parameter, we specify the maximum number of iterations for each single run (here, `300`). Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached. However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for `max_iter`. One way to deal with convergence problems is to choose larger values for `tol`, which is a parameter that controls the tolerance with regard to the changes in the within-cluster SSE to declare convergence. In the preceding code, we chose a tolerance of `1e-04` (=0.0001).

A problem with k-means is that one or more clusters can be empty. Note that this problem does not exist for k-medoids or fuzzy C-means, an algorithm that we will discuss later in this section.

However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the example that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.

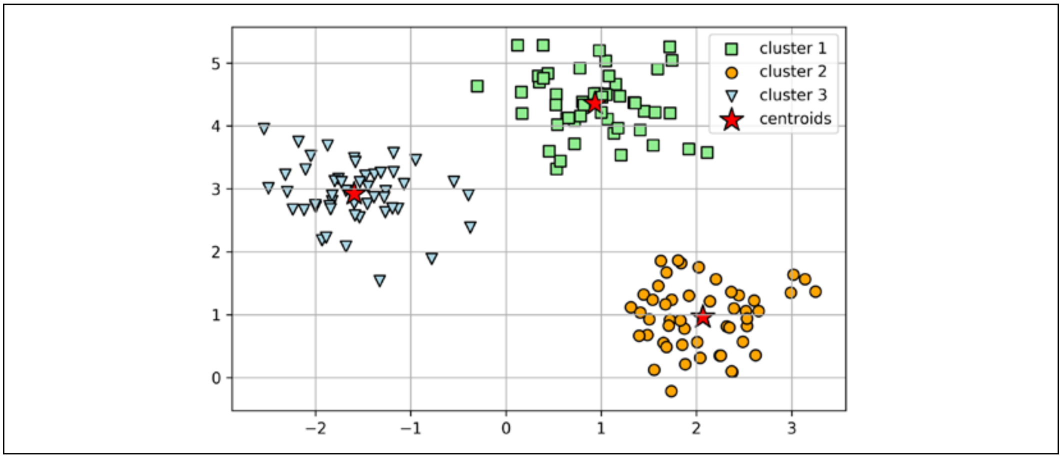
Feature scaling

When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

Having predicted the cluster labels, `y_km`, and discussed some of the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the `cluster_centers_` attribute of the fitted `KMeans` object:

```
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             marker='s', edgecolor='black',
...             label='Cluster 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50, c='orange',
...             marker='o', edgecolor='black',
...             label='Cluster 2')
>>> plt.scatter(X[y_km == 2, 0],
...             X[y_km == 2, 1],
...             s=50, c='lightblue',
...             marker='v', edgecolor='black',
...             label='Cluster 3')
>>> plt.scatter(km.cluster_centers_[:, 0],
...             km.cluster_centers_[:, 1],
...             s=250, marker='*',
...             c='red', edgecolor='black',
...             label='Centroids')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

In the following scatterplot, you can see that k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset:



Although k-means worked well on this toy dataset, we will highlight another drawback of k-means: we have to specify the number of clusters, *k*, *a priori*. The number of clusters to choose may not always be so obvious in real-world applications, especially if we are working with a higher dimensional dataset that cannot be visualized. The other properties of k-means are that clusters do not overlap and are not hierarchical, and we also assume that there is at least one item in each cluster. Later in this chapter, we will encounter different types of clustering algorithms, hierarchical and density-based clustering. Neither type of algorithm requires us to specify the number of clusters upfront or assume spherical structures in our dataset.

In the next subsection, we will cover a popular variant of the classic k-means algorithm called **k-means++**. While it doesn't address those assumptions and drawbacks of k-means that were discussed in the previous paragraph, it can greatly improve the clustering results through more clever seeding of the initial cluster centers.

A smarter way of placing the initial cluster centroids using k-means++

So far, we have discussed the classic k-means algorithm, which uses a random seed to place the initial centroids, which can sometimes result in bad clusterings or slow convergence if the initial centroids are chosen poorly. One way to address this issue is to run the k-means algorithm multiple times on a dataset and choose the best performing model in terms of the SSE.

Another strategy is to place the initial centroids far away from each other via the k-means++ algorithm, which leads to better and more consistent results than the classic k-means (*k-means++: The Advantages of Careful Seeding, D. Arthur* and *S. Vassilvitskii* in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027-1035. *Society for Industrial and Applied Mathematics, 2007*).

The initialization in k-means++ can be summarized as follows:

1. Initialize an empty set, **M**, to store the *k* centroids being selected.
2. Randomly choose the first centroid, $\boldsymbol{\mu}^{(j)}$, from the input examples and assign it to **M**.
3. For each example, $\boldsymbol{x}^{(i)}$, that is not in **M**, find the minimum squared distance, $d\left(\boldsymbol{x}^{(i)}, \mathbf{M}\right)^2$, to any of the centroids in **M**.
4. To randomly select the next centroid, $\boldsymbol{\mu}^{(p)}$, use a weighted probability distribution equal to $\frac{d\left(\boldsymbol{\mu}^{(p)},\ \mathbf{M}\right)^2}{\sum_i d(\boldsymbol{x}^{(i)},\ \mathbf{M})^2}$.
5. Repeat steps 2 and 3 until *k* centroids are chosen.
6. Proceed with the classic k-means algorithm.

To use k-means++ with scikit-learn's `KMeans` object, we just need to set the `init` parameter to `'k-means++'`. In fact, `'k-means++'` is the default argument to the `init` parameter, which is strongly recommended in practice. The only reason we didn't use it in the previous example was to not introduce too many concepts all at once. The rest of this section on k-means will use k-means++, but you are encouraged to experiment more with the two different approaches (classic k-means via `init='random'` versus `k-means++` via `init='k-means++'`) for placing the initial cluster centroids.

Hard versus soft clustering

Hard clustering describes a family of algorithms where each example in a dataset is assigned to exactly one cluster, as in the k-means and k-means++ algorithms that we discussed earlier in this chapter. In contrast, algorithms for **soft clustering** (sometimes also called **fuzzy clustering**) assign an example to one or more clusters. A popular example of soft clustering is the **fuzzy C-means** (**FCM**) algorithm (also called **soft k-means** or **fuzzy k-means**). The original idea goes back to the 1970s, when Joseph C. Dunn first proposed an early version of fuzzy clustering to improve k-means (*A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters, J. C. Dunn, 1973*). Almost a decade later, James C. Bedzek published his work on the improvement of the fuzzy clustering algorithm, which is now known as the FCM algorithm (*Pattern Recognition with Fuzzy Objective Function Algorithms, J. C. Bezdek, Springer Science+Business Media, 2013*).

The FCM procedure is very similar to k-means. However, we replace the hard cluster assignment with probabilities for each point belonging to each cluster. In k-means, we could express the cluster membership of an example, x, with a sparse vector of binary values:

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 1 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0 \end{bmatrix}$$

Here, the index position with value 1 indicates the cluster centroid, $\boldsymbol{\mu}^{(j)}$, that the example is assigned to (assuming $k = 3$, $j \in \{1, 2, 3\}$). In contrast, a membership vector in FCM could be represented as follows:

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0.05 \end{bmatrix}$$

Here, each value falls in the range [0, 1] and represents a probability of membership of the respective cluster centroid. The sum of the memberships for a given example is equal to 1. As with the k-means algorithm, we can summarize the FCM algorithm in four key steps:

1. Specify the number of k centroids and randomly assign the cluster memberships for each point.
2. Compute the cluster centroids, $\boldsymbol{\mu}^{(j)}, j \in \{1, \dots, k\}$.
3. Update the cluster memberships for each point.
4. Repeat steps 2 and 3 until the membership coefficients do not change or a user-defined tolerance or maximum number of iterations is reached.

The objective function of FCM—we abbreviate it as J_m—looks very similar to the within-cluster SSE that we minimize in k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)^m} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

However, note that the membership indicator, $w^{(i,j)}$, is not a binary value as in k-means ($w^{(i,j)} \in \{0, 1\}$), but a real value that denotes the cluster membership probability ($w^{(i,j)} \in [0, 1]$). You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent m, any number greater than or equal to one (typically m = 2), is the so-called **fuzziness coefficient** (or simply **fuzzifier**), which controls the degree of *fuzziness*.

The larger the value of *m*, the smaller the cluster membership, $w^{(i,j)}$, becomes, which leads to fuzzier clusters. The cluster membership probability itself is calculated as follows:

$$w^{(i,j)} = \left[\sum_{c=1}^{k}\left(\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(c)}\right\|_2}\right)^{\frac{2}{m-1}}\right]^{-1}$$

For example, if we chose three cluster centers, as in the previous k-means example, we could calculate the membership of $\boldsymbol{x}^{(i)}$ belonging to the $\boldsymbol{\mu}^{(j)}$ cluster as follows:

$$w^{(i,j)} = \left[\left(\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)}\right\|_2}\right)^{\frac{2}{m-1}} + \left(\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)}\right\|_2}\right)^{\frac{2}{m-1}} + \left(\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2}{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)}\right\|_2}\right)^{\frac{2}{m-1}}\right]^{-1}$$

The center, $\boldsymbol{\mu}^{(j)}$, of a cluster itself is calculated as the mean of all examples weighted by the degree to which each example belongs to that cluster ($w^{(i,j)^m}$):

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{(i,j)^m}\boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{(i,j)^m}}$$

Just by looking at the equation to calculate the cluster memberships, we can say that each iteration in FCM is more expensive than an iteration in k-means. On the other hand, FCM typically requires fewer iterations overall to reach convergence. Unfortunately, the FCM algorithm is currently not implemented in scikit-learn. However, it has been found, in practice, that both k-means and FCM produce very similar clustering outputs, as described in a study (*Comparative Analysis of k-means and Fuzzy C-Means Algorithms*, *S. Ghosh*, and *S. K. Dubey*, *IJACSA*, 4: 35–38, *2013*).

Using the elbow method to find the optimal number of clusters

One of the main challenges in unsupervised learning is that we do not know the definitive answer. We don't have the ground truth class labels in our dataset that allow us to apply the techniques that we used in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, in order to evaluate the performance of a supervised model. Thus, to quantify the quality of clustering, we need to use intrinsic metrics—such as the within-cluster SSE (distortion)—to compare the performance of different k-means clusterings.

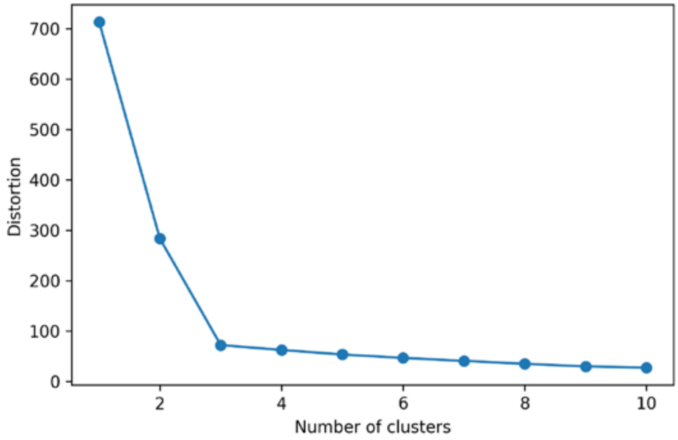
Conveniently, we don't need to compute the within-cluster SSE explicitly when we are using scikit-learn, as it is already accessible via the `inertia_` attribute after fitting a `KMeans` model:

```
>>> print('Distortion: %.2f' % km.inertia_)
Distortion: 72.48
```

Based on the within-cluster SSE, we can use a graphical tool, the so-called **elbow method**, to estimate the optimal number of clusters, *k*, for a given task. We can say that if *k* increases, the distortion will decrease. This is because the examples will be closer to the centroids they are assigned to. The idea behind the elbow method is to identify the value of *k* where the distortion begins to increase most rapidly, which will become clearer if we plot the distortion for different values of *k*:

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
...     km.fit(X)
...     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in the following plot, the *elbow* is located at *k* = 3, so this is evidence that *k* = 3 is indeed a good choice for this dataset:



Quantifying the quality of clustering via silhouette plots

Another intrinsic metric to evaluate the quality of a clustering is **silhouette analysis**, which can also be applied to clustering algorithms other than k-means, which we will discuss later in this chapter. Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the examples in the clusters are. To calculate the **silhouette coefficient** of a single example in our dataset, we can apply the following three steps:

1. Calculate the **cluster cohesion**, $a^{(i)}$, as the average distance between an example, $\boldsymbol{x}^{(i)}$, and all other points in the same cluster.
2. Calculate the **cluster separation**, $b^{(i)}$, from the next closest cluster as the average distance between the example, $\boldsymbol{x}^{(i)}$, and all examples in the nearest cluster.
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

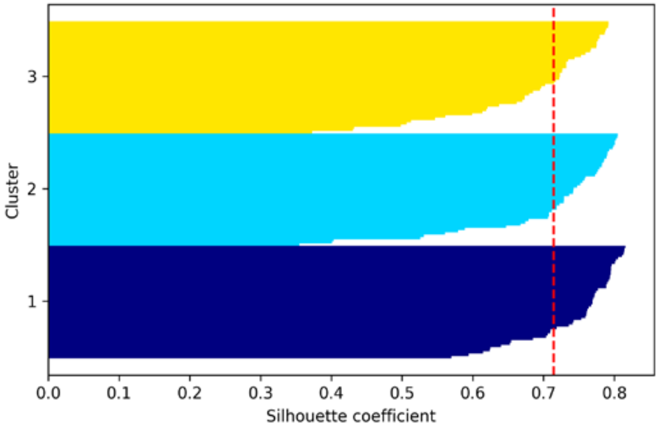
$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{max\{ b^{(i)}, a^{(i)}\}}$$

The silhouette coefficient is bounded in the range –1 to 1. Based on the preceding equation, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} \gg a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar an example is from other clusters, and $a^{(i)}$ tells us how similar it is to the other examples in its own cluster.

The silhouette coefficient is available as `silhouette_samples` from scikit-learn's `metric` module, and optionally, the `silhouette_scores` function can be imported for convenience. The `silhouette_scores` function calculates the average silhouette coefficient across all examples, which is equivalent to `numpy.mean(silhouette_samples(...))`. By executing the following code, we will now create a plot of the silhouette coefficients for a k-means clustering with *k* = 3:

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

Through a visual inspection of the silhouette plot, we can quickly scrutinize the sizes of the different clusters and identify clusters that contain *outliers*:



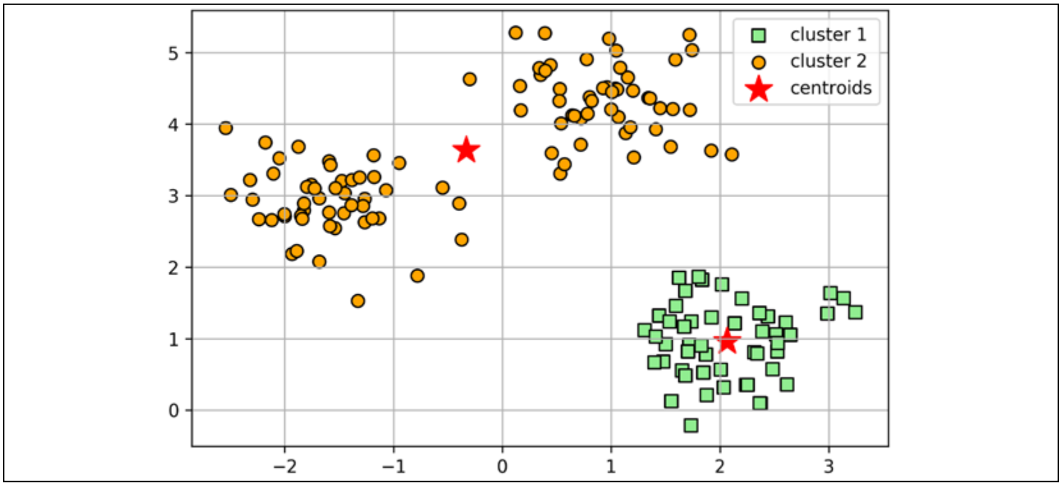
However, as you can see in the preceding silhouette plot, the silhouette coefficients are not even close to **0**, which is, in this case, an indicator of a *good* clustering. Furthermore, to summarize the goodness of our clustering, we added the average silhouette coefficient to the plot (dotted line).

To see what a silhouette plot looks like for a relatively *bad* clustering, let's seed the k-means algorithm with only two centroids:

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             edgecolor='black',
...             marker='s',
...             label='Cluster 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50,
...             c='orange',
...             edgecolor='black',
...             marker='o',
...             label='Cluster 2')
>>> plt.scatter(km.cluster_centers_[:, 0],
...             km.cluster_centers_[:, 1],
...             s=250,
...             marker='*',
...             c='red',
...             label='Centroids')
>>> plt.legend()
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in the resulting plot, one of the centroids falls between two of the three spherical groupings of the input data.

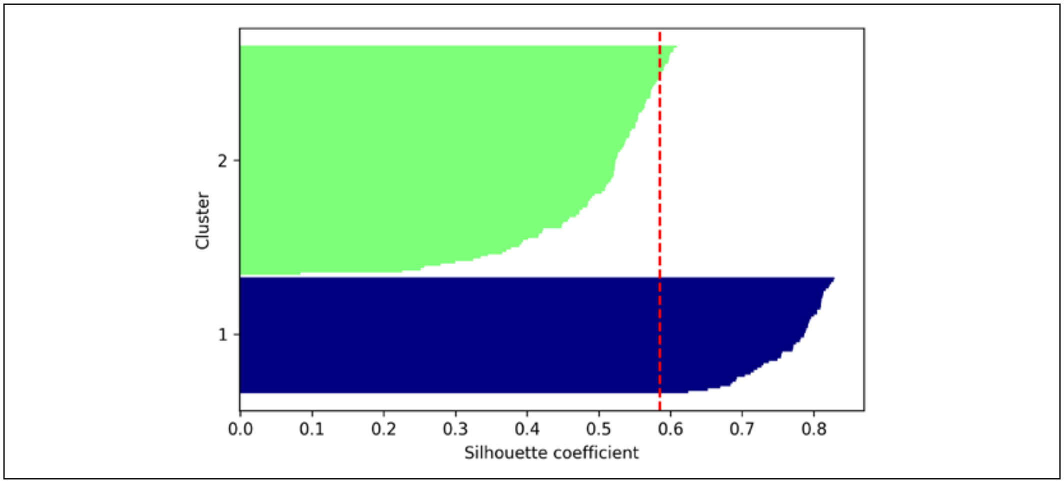
Although the clustering does not look completely terrible, it is suboptimal:



Please keep in mind that we typically do not have the luxury of visualizing datasets in two-dimensional scatterplots in real-world problems, since we typically work with data in higher dimensions. So, next, we will create the silhouette plot to evaluate the results:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                      y_km,
...                                      metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in the resulting plot, the silhouettes now have visibly different lengths and widths, which is evidence of a relatively *bad* or at least *suboptimal* clustering:



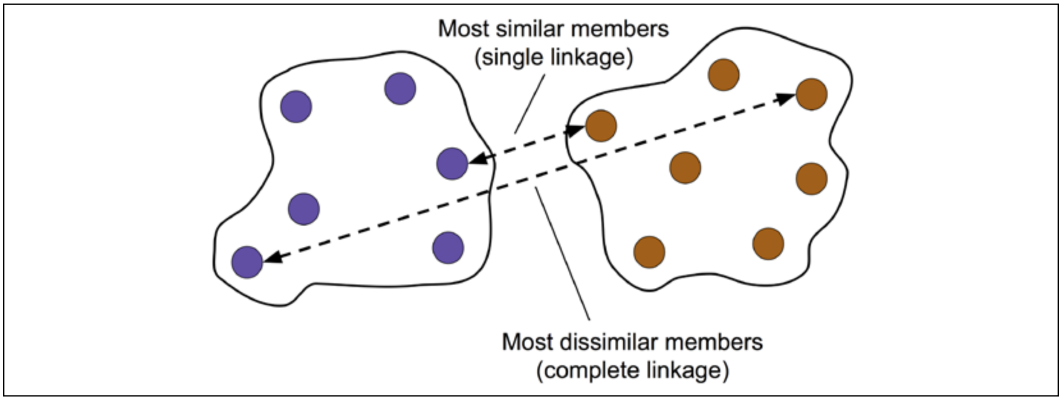
Organizing clusters as a hierarchical tree

In this section, we will take a look at an alternative approach to prototype-based clustering: **hierarchical clustering**. One advantage of the hierarchical clustering algorithm is that it allows us to plot **dendrograms** (visualizations of a binary hierarchical clustering), which can help with the interpretation of the results by creating meaningful taxonomies. Another advantage of this hierarchical approach is that we do not need to specify the number of clusters upfront.

The two main approaches to hierarchical clustering are **agglomerative** and **divisive** hierarchical clustering. In divisive hierarchical clustering, we start with one cluster that encompasses the complete dataset, and we iteratively split the cluster into smaller clusters until each cluster only contains one example. In this section, we will focus on agglomerative clustering, which takes the opposite approach. We start with each example as an individual cluster and merge the closest pairs of clusters until only one cluster remains.

Grouping clusters in bottom-up fashion

The two standard algorithms for agglomerative hierarchical clustering are **single linkage** and **complete linkage**. Using single linkage, we compute the distances between the most similar members for each pair of clusters and merge the two clusters for which the distance between the most similar members is the smallest. The complete linkage approach is similar to single linkage but, instead of comparing the most similar members in each pair of clusters, we compare the most dissimilar members to perform the merge. This is shown in the following diagram:



Alternative types of linkages

Other commonly used algorithms for agglomerative hierarchical clustering include **average linkage** and **Ward's linkage**. In average linkage, we merge the cluster pairs based on the minimum average distances between all group members in the two clusters. In Ward's linkage, the two clusters that lead to the minimum increase of the total within-cluster SSE are merged.

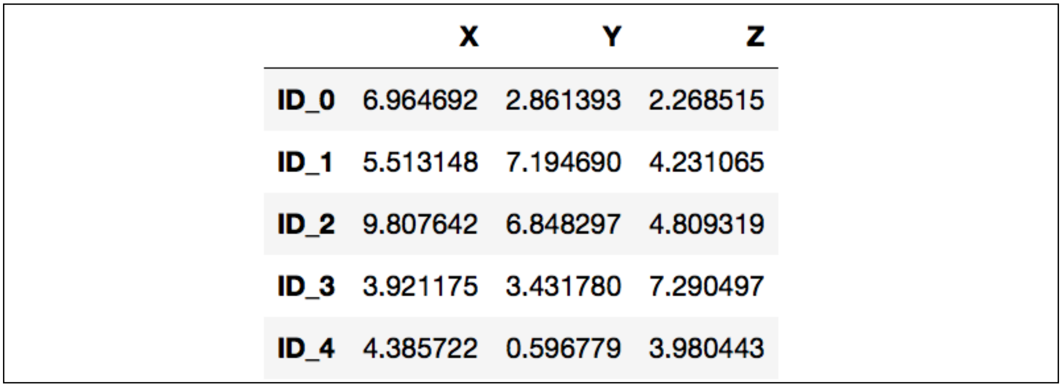
In this section, we will focus on agglomerative clustering using the complete linkage approach. Hierarchical complete linkage clustering is an iterative procedure that can be summarized by the following steps:

1. Compute the distance matrix of all examples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
4. Update the similarity matrix.
5. Repeat steps 2-4 until one single cluster remains.

Next, we will discuss how to compute the distance matrix (step 1). But first, let's generate a random data sample to work with. The rows represent different observations (IDs 0-4), and the columns are the different features (X, Y, Z) of those examples:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random_sample([5, 3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

After executing the preceding code, we should now see the following data frame containing the randomly generated examples:



	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

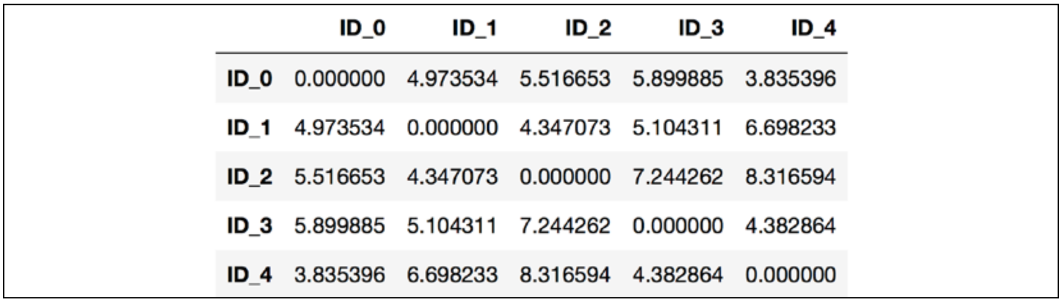
Performing hierarchical clustering on a distance matrix

To calculate the distance matrix as input for the hierarchical clustering algorithm, we will use the pdist function from SciPy's spatial.distance submodule:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                         pdist(df, metric='euclidean')),
...                         columns=labels, index=labels)
>>> row_dist
```

Using the preceding code, we calculated the Euclidean distance between each pair of input examples in our dataset based on the features X, Y, and Z.

We provided the condensed distance matrix—returned by `pdist`—as input to the `squareform` function to create a symmetrical matrix of the pair-wise distances, as shown here:



	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Next, we will apply the complete linkage agglomeration to our clusters using the `linkage` function from SciPy's `cluster.hierarchy` submodule, which returns a so-called **linkage matrix**.

However, before we call the `linkage` function, let's take a careful look at the function documentation:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.

  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

  metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

  Returns:
  Z : ndarray
    The hierarchical clustering encoded as a linkage matrix.
[...]
```

Based on the function description, we understand that we can use a condensed distance matrix (upper triangular) from the `pdist` function as an input attribute. Alternatively, we could also provide the initial data array and use the `'euclidean'` metric as a function argument in `linkage`. However, we should not use the `squareform` distance matrix that we defined earlier, since it would yield different distance values than expected. To sum it up, the three possible scenarios are listed here:

- **Incorrect approach**: Using the `squareform` distance matrix as shown in the following code snippet leads to incorrect results:

  ```
  >>> row_clusters = linkage(row_dist,
  ...                        method='complete',
  ...                        metric='euclidean')
  ```

- **Correct approach**: Using the condensed distance matrix as shown in the following code example yields the correct linkage matrix:

  ```
  >>> row_clusters = linkage(pdist(df, metric='euclidean'),
  ...                        method='complete')
  ```

- **Correct approach**: Using the complete input example matrix (the so-called design matrix) as shown in the following code snippet also leads to a correct linkage matrix similar to the preceding approach:

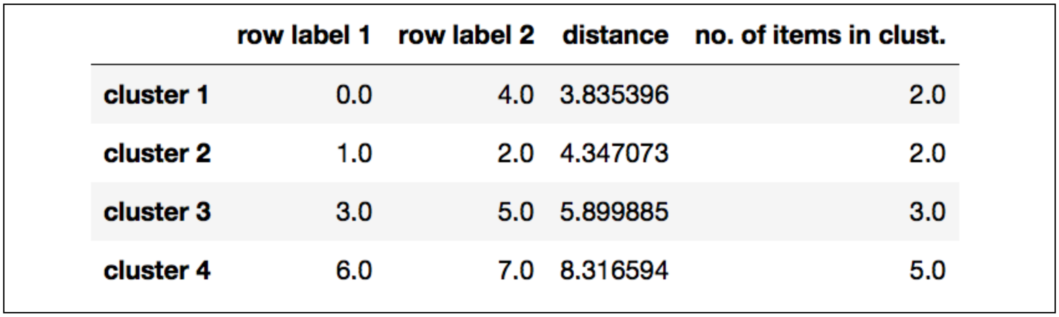
  ```
  >>> row_clusters = linkage(df.values,
  ...                        method='complete',
  ...                        metric='euclidean')
  ```

To take a closer look at the clustering results, we can turn those results into a pandas `DataFrame` (best viewed in a Jupyter Notebook) as follows:

```
>>> pd.DataFrame(row_clusters,
...              columns=['row label 1',
...                       'row label 2',
...                       'distance',
...                       'no. of items in clust.'],
...              index=['cluster %d' % (i + 1) for i in
...                     range(row_clusters.shape[0])])
```

As shown in the following screenshot, the linkage matrix consists of several rows where each row represents one merge. The first and second columns denote the most dissimilar members in each cluster, and the third column reports the distance between those members.

The last column returns the count of the members in each cluster:

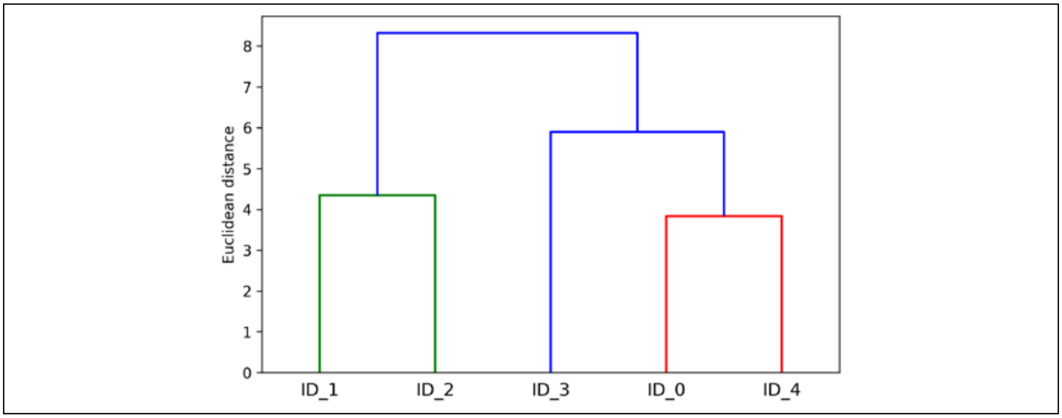


	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0

Now that we have computed the linkage matrix, we can visualize the results in the form of a dendrogram:

```
>>> from scipy.cluster.hierarchy import dendrogram
>>> # make dendrogram black (part 1/2)
>>> # from scipy.cluster.hierarchy import set_link_color_palette
>>> # set_link_color_palette(['black'])
>>> row_dendr = dendrogram(row_clusters,
...                        labels=labels,
...                        # make dendrogram black (part 2/2)
...                        # color_threshold=np.inf
...                        )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

If you are executing the preceding code or reading an e-book version of this book, you will notice that the branches in the resulting dendrogram are shown in different colors. The color scheme is derived from a list of Matplotlib colors that are cycled for the distance thresholds in the dendrogram. For example, to display the dendrograms in black, you can uncomment the respective sections that were inserted in the preceding code:



Such a dendrogram summarizes the different clusters that were formed during the agglomerative hierarchical clustering; for example, you can see that the examples `ID_0` and `ID_4`, followed by `ID_1` and `ID_2`, are the most similar ones based on the Euclidean distance metric.

Attaching dendrograms to a heat map

In practical applications, hierarchical clustering dendrograms are often used in combination with a **heat map**, which allows us to represent the individual values in the data array or matrix containing our training examples with a color code. In this section, we will discuss how to attach a dendrogram to a heat map plot and order the rows in the heat map correspondingly.

However, attaching a dendrogram to a heat map can be a little bit tricky, so let's go through this procedure step by step:

1. We create a new `figure` object and define the *x* axis position, *y* axis position, width, and height of the dendrogram via the `add_axes` attribute. Furthermore, we rotate the dendrogram 90 degrees counter-clockwise. The code is as follows:

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>>> axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
>>> row_dendr = dendrogram(row_clusters,
...                        orientation='left')
>>> # note: for matplotlib < v1.5.1, please use
>>> # orientation='right'
```

2. Next, we reorder the data in our initial `DataFrame` according to the clustering labels that can be accessed from the `dendrogram` object, which is essentially a Python dictionary, via the `leaves` key. The code is as follows:

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][::-1]]
```

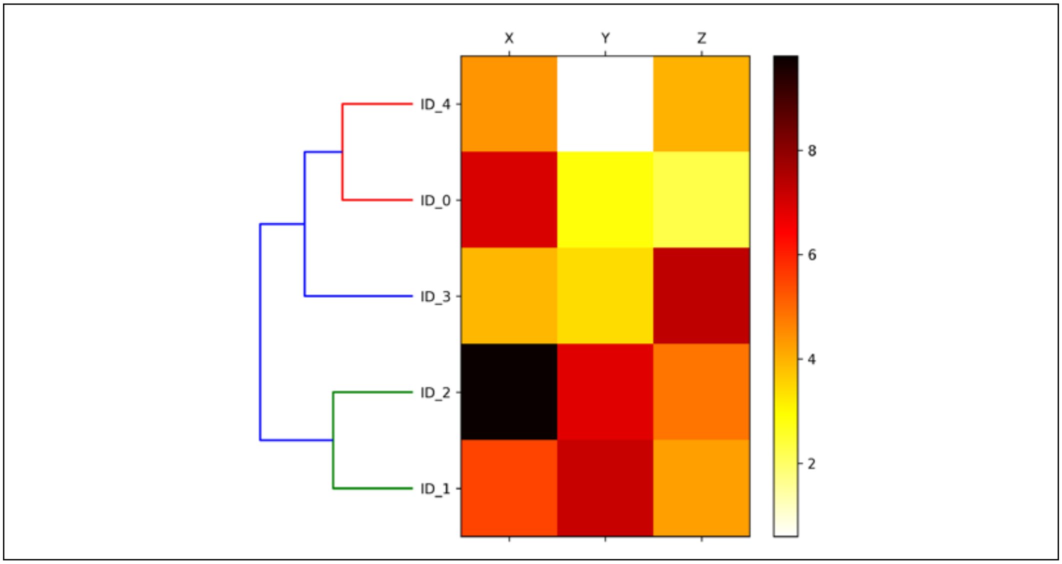
3. Now, we construct the heat map from the reordered `DataFrame` and position it next to the dendrogram:

```
>>> axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
>>> cax = axm.matshow(df_rowclust,
...                   interpolation='nearest',
...                   cmap='hot_r')
```

4. Finally, we modify the aesthetics of the dendrogram by removing the axis ticks and hiding the axis spines. Also, we add a color bar and assign the feature and data record names to the *x* and *y* axis tick labels, respectively:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

After following the previous steps, the heat map should be displayed with the dendrogram attached:



As you can see, the order of rows in the heat map reflects the clustering of the examples in the dendrogram. In addition to a simple dendrogram, the color-coded values of each example and feature in the heat map provide us with a nice summary of the dataset.

Applying agglomerative clustering via scikit-learn

In the previous subsection, you saw how to perform agglomerative hierarchical clustering using SciPy. However, there is also an `AgglomerativeClustering` implementation in scikit-learn, which allows us to choose the number of clusters that we want to return. This is useful if we want to prune the hierarchical cluster tree. By setting the `n_cluster` parameter to `3`, we will now cluster the input examples into three groups using the same complete linkage approach based on the Euclidean distance metric, as before:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Cluster labels: %s' % labels)
Cluster labels: [1 0 0 2 1]
```

Looking at the predicted cluster labels, we can see that the first and the fifth examples (`ID_0` and `ID_4`) were assigned to one cluster (label `1`), and the examples `ID_1` and `ID_2` were assigned to a second cluster (label `0`). The example `ID_3` was put into its own cluster (label `2`). Overall, the results are consistent with the results that we observed in the dendrogram. We should note, though, that `ID_3` is more similar to `ID_4` and `ID_0` than to `ID_1` and `ID_2`, as shown in the preceding dendrogram figure; this is not clear from scikit-learn's clustering results. Let's now rerun the `AgglomerativeClustering` using `n_cluster=2` in the following code snippet:

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Cluster labels: %s' % labels)
Cluster labels: [0 1 1 0 0]
```

As you can see, in this *pruned* clustering hierarchy, label `ID_3` was assigned to the same cluster as `ID_0` and `ID_4`, as expected.

Locating regions of high density via DBSCAN

Although we can't cover the vast amount of different clustering algorithms in this chapter, let's at least include one more approach to clustering: **density-based spatial clustering of applications with noise** (**DBSCAN**), which does not make assumptions about spherical clusters like k-means, nor does it partition the dataset into hierarchies that require a manual cut-off point. As its name implies, density-based clustering assigns cluster labels based on dense regions of points. In DBSCAN, the notion of density is defined as the number of points within a specified radius, ε.

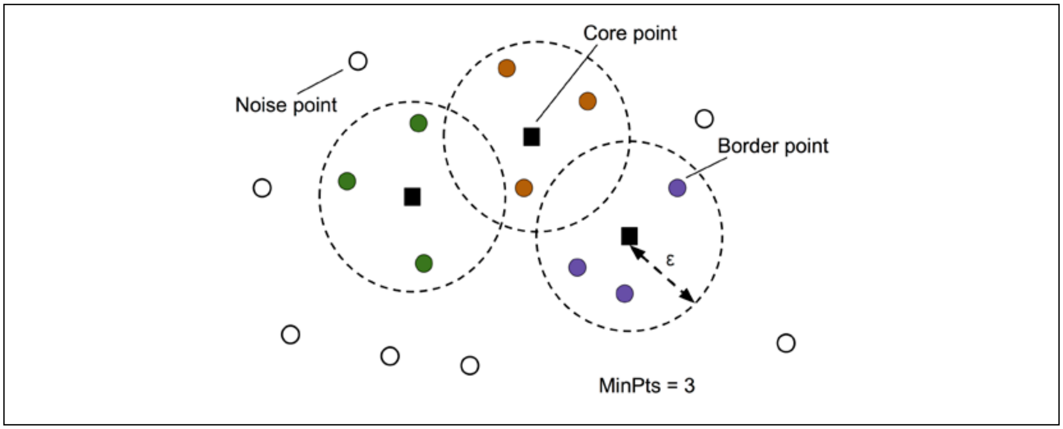
According to the DBSCAN algorithm, a special label is assigned to each example (data point) using the following criteria:

- A point is considered a **core point** if at least a specified number (MinPts) of neighboring points fall within the specified radius, ε.
- A **border point** is a point that has fewer neighbors than MinPts within ε, but lies within the ε radius of a core point.
- All other points that are neither core nor border points are considered **noise points**.

After labeling the points as core, border, or noise, the DBSCAN algorithm can be summarized in two simple steps:

1. Form a separate cluster for each core point or connected group of core points. (Core points are connected if they are no farther away than ε.)
2. Assign each border point to the cluster of its corresponding core point.

To get a better understanding of what the result of DBSCAN can look like, before jumping to the implementation, let's summarize what we have just learned about core points, border points, and noise points in the following figure:

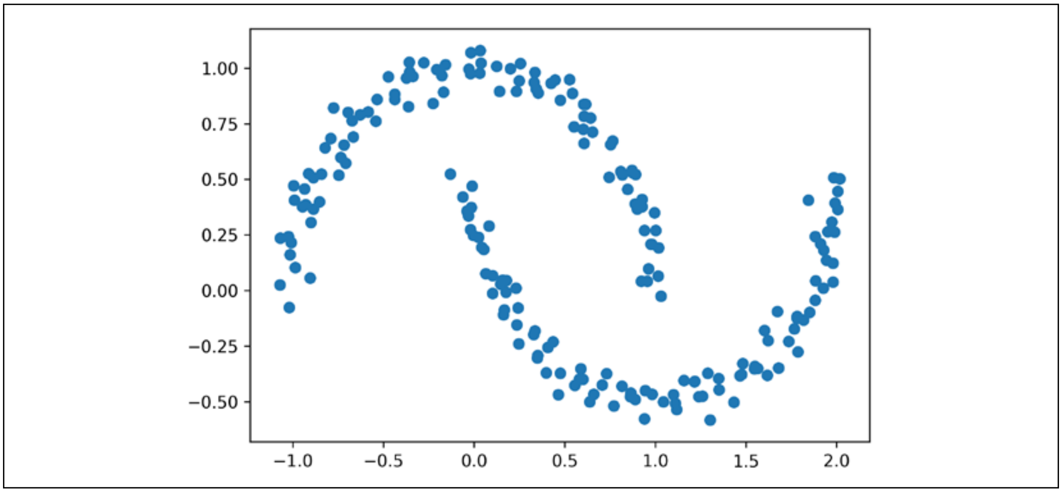


One of the main advantages of using DBSCAN is that it does not assume that the clusters have a spherical shape as in k-means. Furthermore, DBSCAN is different from k-means and hierarchical clustering in that it doesn't necessarily assign each point to a cluster but is capable of removing noise points.

For a more illustrative example, let's create a new dataset of half-moon-shaped structures to compare k-means clustering, hierarchical clustering, and DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                   noise=0.05,
...                   random_state=0)
>>> plt.scatter(X[:, 0], X[:, 1])
>>> plt.tight_layout()
>>> plt.show()
```

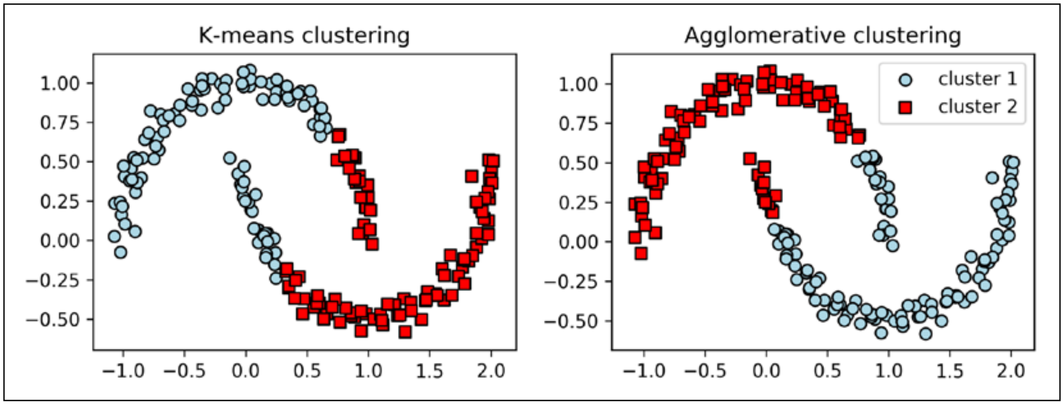
As you can see in the resulting plot, there are two visible, half-moon-shaped groups consisting of 100 examples (data points) each:



We will start by using the k-means algorithm and complete linkage clustering to see if one of those previously discussed clustering algorithms can successfully identify the half-moon shapes as separate clusters. The code is as follows:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='cluster 1')
>>> ax1.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac == 0, 0],
...             X[y_ac == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='Cluster 1')
>>> ax2.scatter(X[y_ac == 1, 0],
...             X[y_ac == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='Cluster 2')
>>> ax2.set_title('Agglomerative clustering')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

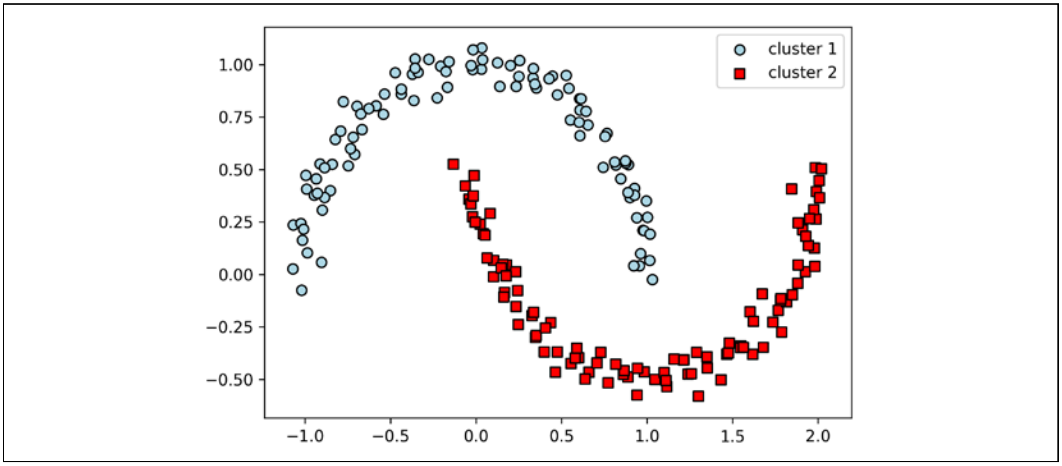
Based on the visualized clustering results, we can see that the k-means algorithm was unable to separate the two clusters, and also, the hierarchical clustering algorithm was challenged by those complex shapes:



Finally, let's try the DBSCAN algorithm on this dataset to see if it can find the two half-moon-shaped clusters using a density-based approach:

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db == 0, 0],
...             X[y_db == 0, 1],
...                 c='lightblue',
...                 edgecolor='black',
...                 marker='o',
...                 s=40,
...                 label='Cluster 1')
>>> plt.scatter(X[y_db == 1, 0],
...                 X[y_db == 1, 1],
...                 c='red',
...                 edgecolor='black',
...                 marker='s',
...                 s=40,
...                 label='Cluster 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

The DBSCAN algorithm can successfully detect the half-moon shapes, which highlights one of the strengths of DBSCAN – clustering data of arbitrary shapes:



However, we should also note some of the disadvantages of DBSCAN. With an increasing number of features in our dataset—assuming a fixed number of training examples—the negative effect of the **curse of dimensionality** increases. This is especially a problem if we are using the Euclidean distance metric. However, the problem of the curse of dimensionality is not unique to DBSCAN: it also affects other clustering algorithms that use the Euclidean distance metric, for example, k-means and hierarchical clustering algorithms. In addition, we have two hyperparameters in DBSCAN (MinPts and ε) that need to be optimized to yield good clustering results. Finding a good combination of MinPts and ε can be problematic if the density differences in the dataset are relatively large.

Graph-based clustering

So far, we have seen three of the most fundamental categories of clustering algorithms: prototype-based clustering with k-means, agglomerative hierarchical clustering, and density-based clustering via DBSCAN. However, there is also a fourth class of more advanced clustering algorithms that we have not covered in this chapter: **graph-based clustering**. Probably the most prominent members of the graph-based clustering family are the **spectral clustering** algorithms.

Although there are many different implementations of spectral clustering, what they all have in common is that they use the eigenvectors of a similarity or distance matrix to derive the cluster relationships. Since spectral clustering is beyond the scope of this book, you can read the excellent tutorial by Ulrike von Luxburg to learn more about this topic (*A tutorial on spectral clustering*, *U. Von Luxburg*, *Statistics and Computing*, 17(4): 395–416, *2007*). It is freely available from arXiv at `http://arxiv.org/pdf/0711.0189v1.pdf`.

Note that, in practice, it is not always obvious which clustering algorithm will perform best on a given dataset, especially if the data comes in multiple dimensions that make it hard or impossible to visualize. Furthermore, it is important to emphasize that a successful clustering does not only depend on the algorithm and its hyperparameters; rather, the choice of an appropriate distance metric and the use of domain knowledge that can help to guide the experimental setup can be even more important.

In the context of the curse of dimensionality, it is thus common practice to apply dimensionality reduction techniques prior to performing clustering. Such dimensionality reduction techniques for unsupervised datasets include principal component analysis and radial basis function kernel principal component analysis, which we covered in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Also, it is particularly common to compress datasets down to two-dimensional subspaces, which allows us to visualize the clusters and assigned labels using two-dimensional scatterplots, which are particularly helpful for evaluating the results.

Summary

In this chapter, you learned about three different clustering algorithms that can help us with the discovery of hidden structures or information in data. We started this chapter with a prototype-based approach, k-means, which clusters examples into spherical shapes based on a specified number of cluster centroids. Since clustering is an unsupervised method, we do not enjoy the luxury of ground truth labels to evaluate the performance of a model. Thus, we used intrinsic performance metrics, such as the elbow method or silhouette analysis, as an attempt to quantify the quality of clustering.

We then looked at a different approach to clustering: agglomerative hierarchical clustering. Hierarchical clustering does not require specifying the number of clusters upfront, and the result can be visualized in a dendrogram representation, which can help with the interpretation of the results. The last clustering algorithm that we covered in this chapter was DBSCAN, an algorithm that groups points based on local densities and is capable of handling outliers and identifying non-globular shapes.

After this excursion into the field of unsupervised learning, it is now about time to introduce some of the most exciting machine learning algorithms for supervised learning: multilayer artificial neural networks. After their recent resurgence, neural networks are once again the hottest topic in machine learning research. Thanks to recently developed deep learning algorithms, neural networks are considered state-of-the-art for many complex tasks such as image classification and speech recognition. In *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*, we will construct our own multilayer neural network. In *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, we will work with the TensorFlow library, which specializes in training neural network models with multiple layers very efficiently by utilizing graphics processing units.

12

Implementing a Multilayer Artificial Neural Network from Scratch

As you may know, deep learning is getting a lot of attention from the press and is without any doubt the hottest topic in the machine learning field. Deep learning can be understood as a subfield of machine learning that is concerned with training artificial neural networks (NNs) with many layers efficiently. In this chapter, you will learn the basic concepts of artificial NNs so that you are well equipped for the following chapters, which will introduce advanced Python-based deep learning libraries and **deep neural network** (**DNN**) architectures that are particularly well suited for image and text analyses.

The topics that we will cover in this chapter are as follows:

- Gaining a conceptual understanding of multilayer NNs
- Implementing the fundamental backpropagation algorithm for NN training from scratch
- Training a basic multilayer NN for image classification

Modeling complex functions with artificial neural networks

At the beginning of this book, we started our journey through machine learning algorithms with artificial neurons in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. Artificial neurons represent the building blocks of the multilayer artificial NNs that we will discuss in this chapter.

The basic concept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problem tasks. Although artificial NNs have gained a lot of popularity in recent years, early studies of NNs go back to the 1940s when Warren McCulloch and Walter Pitts first described how neurons could work. (*A logical calculus of the ideas immanent in nervous activity*, *W. S. McCulloch* and *W. Pitts*. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.)

However, in the decades that followed the first implementation of the **McCulloch-Pitts neuron** model—Rosenblatt's perceptron in the 1950s — many researchers and machine learning practitioners slowly began to lose interest in NNs since no one had a good solution for training an NN with multiple layers. Eventually, interest in NNs was rekindled in 1986 when D.E. Rumelhart, G.E. Hinton, and R.J. Williams were involved in the (re)discovery and popularization of the backpropagation algorithm to train NNs more efficiently, which we will discuss in more detail later in this chapter (*Learning representations by back-propagating errors*, *D. E. Rumelhart*, *G. E. Hinton*, *R. J. Williams*, *Nature*, 323 (6088): 533–536, *1986*). Readers who are interested in the history of **artificial intelligence** (**AI**), machine learning, and NNs are also encouraged to read the Wikipedia article on the so-called *AI winters*, which are the periods of time where a large portion of the research community lost interest in the study of NNs (`https://en.wikipedia.org/wiki/AI_winter`).

However, NNs are more popular today than ever thanks to the many major breakthroughs that have been made in the previous decade, which resulted in what we now call deep learning algorithms and architectures—NNs that are composed of many layers. NNs are a hot topic not only in academic research but also in big technology companies, such as Facebook, Microsoft, Amazon, Uber, and Google, that invest heavily in artificial NNs and deep learning research.

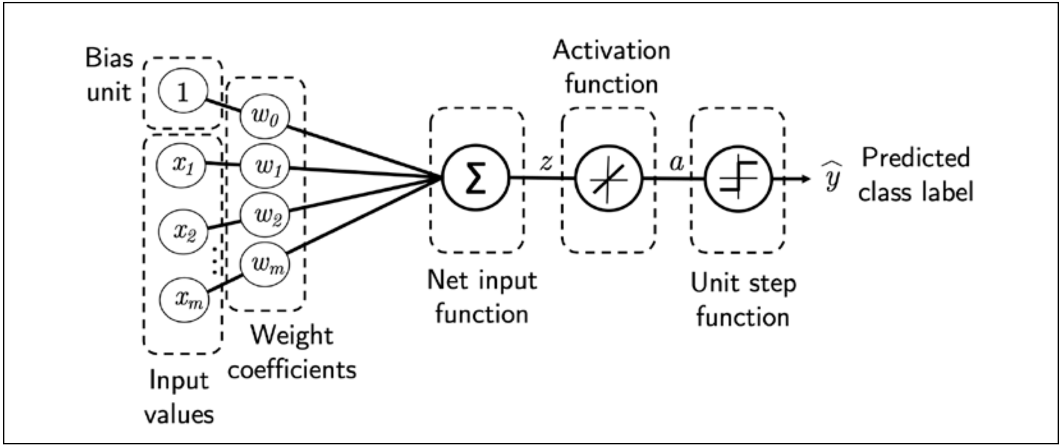
As of today, complex NNs powered by deep learning algorithms are considered the state-of-the-art solutions for complex problem solving such as image and voice recognition. Popular examples of the products in our everyday life that are powered by deep learning are Google's image search and Google Translate—an application for smartphones that can automatically recognize text in images for real-time translation into more than 20 languages.

Many exciting applications of DNNs have been developed at major tech companies and the pharmaceutical industry as listed in the following, non-comprehensive list of examples:

- Facebook's DeepFace for tagging images (*DeepFace: Closing the Gap to Human-Level Performance in Face Verification*, *Y. Taigman*, *M. Yang*, *M. Ranzato*, and *L. Wolf*, *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1701–1708, *2014*)
- Baidu's DeepSpeech, which is able to handle voice queries in Mandarin (*DeepSpeech: Scaling up end-to-end speech recognition*, *A. Hannun*, *C. Case*, *J. Casper*, *B. Catanzaro*, *G. Diamos*, *E. Elsen*, *R. Prenger*, *S. Satheesh*, *S. Sengupta*, *A. Coates*, and *Andrew Y. Ng*, arXiv preprint arXiv:1412.5567, *2014*)
- Google's new language translation service (*Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, arXiv preprint arXiv:1412.5567, *2016*)
- Novel techniques for drug discovery and toxicity prediction (*Toxicity prediction using Deep Learning*, *T. Unterthiner*, *A. Mayr*, *G. Klambauer*, and *S. Hochreiter*, arXiv preprint arXiv:1503.01445, *2015*)
- A mobile application that can detect skin cancer with an accuracy similar to professionally trained dermatologists (*Dermatologist-level classification of skin cancer with deep neural networks*, *A. Esteva*, *B.Kuprel*, *R. A. Novoa*, *J. Ko*, *S. M. Swetter*, *H. M. Blau*, and *S.Thrun*, in *Nature* 542, no. 7639, *2017*, pages 115-118)
- Protein 3D structure prediction from gene sequences (*De novo structure prediction with deep-learning based scoring, R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T.F.G. Green, C. Qin, A. Zidek, A. Nelson, A. Bridgland, H. Penedones, S. Petersen, K. Simonyan, S. Crossan, D.T. Jones, D. Silver, K. Kavukcuoglu, D. Hassabis, and A.W. Senior*, in *Thirteenth Critical Assessment of Techniques for Protein Structure Prediction*, 1-4 December, *2018*)
- Learning how to drive in dense traffic from purely observational data such as camera video streams (*Model-predictive policy learning with uncertainty regularization for driving in dense traffic, M. Henaff, A. Canziani, Y. LeCun, 2019*, in *Conference Proceedings of the International Conference on Learning Representations, ICLR*, 2019)

Single-layer neural network recap

This chapter is all about multilayer NNs, how they work, and how to train them to solve complex problems. However, before we dig deeper into a particular multilayer NN architecture, let's briefly reiterate some of the concepts of single-layer NNs that we introduced in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, namely, the **ADAptive LInear NEuron** (**Adaline**) algorithm, which is shown in the following figure:



In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, we implemented the Adaline algorithm to perform binary classification, and we used the gradient descent optimization algorithm to learn the weight coefficients of the model. In every epoch (pass over the training dataset), we updated the weight vector ***w*** using the following update rule:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad \text{where } \Delta\boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

In other words, we computed the gradient based on the whole training dataset and updated the weights of the model by taking a step into the opposite direction of the gradient $\nabla J(\boldsymbol{w})$. In order to find the optimal weights of the model, we optimized an objective function that we defined as the **sum of squared errors** (**SSE**) cost function $J(\boldsymbol{w})$. Furthermore, we multiplied the gradient by a factor, the **learning rate** η, which we had to choose carefully to balance the speed of learning against the risk of overshooting the global minimum of the cost function.

In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight w_j in the weight vector ***w*** as follows:

$$\frac{\partial}{\partial w_j} J(\boldsymbol{w}) = -\sum_i \left(y^{(i)} - a^{(i)}\right) x_j^{(i)}$$

Here, $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the activation of the neuron, which is a linear function in the special case of Adaline.

Furthermore, we defined the activation function $\phi(\cdot)$ as follows:

$$\phi(z) = z = a$$

Here, the net input, z, is a linear combination of the weights that are connecting the input layer to the output layer:

$$z = \sum_j w_j x_j = \boldsymbol{w}^T \boldsymbol{x}$$

While we used the activation $\phi(z)$ to compute the gradient update, we implemented a threshold function to squash the continuous valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } g(z) \geq \ 0; \\ -1 & \text{otherwise} \end{cases}$$

Single-layer naming convention

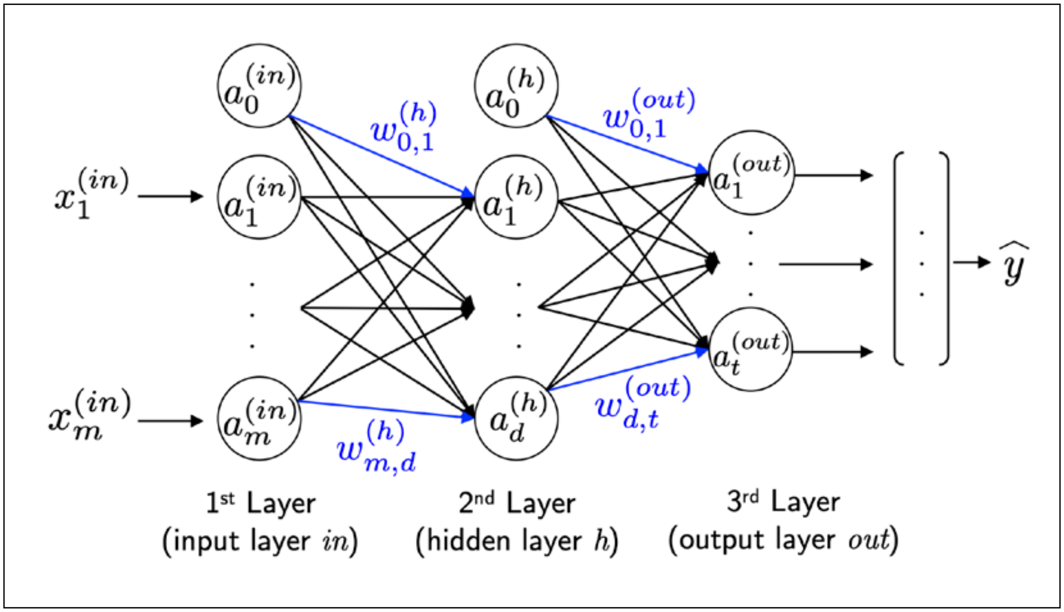
Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Also, we learned about a certain *trick* to accelerate the model learning, the so-called **stochastic gradient descent** (**SGD**) optimization. SGD approximates the cost from a single training sample (online learning) or a small subset of training examples (mini-batch learning). We will make use of this concept later in this chapter when we implement and train a multilayer perceptron (MLP). Apart from faster learning—due to the more frequent weight updates compared to gradient descent—its noisy nature is also regarded as beneficial when training multilayer NNs with nonlinear activation functions, which do not have a convex cost function. Here, the added noise can help to escape local cost minima, but we will discuss this topic in more detail later in this chapter.

Introducing the multilayer neural network architecture

In this section, you will learn how to connect multiple single neurons to a multilayer feedforward NN; this special type of *fully connected* network is also called **MLP**.

The following figure illustrates the concept of an MLP consisting of three layers:



The MLP depicted in the preceding figure has one input layer, one hidden layer, and one output layer. The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer. If such a network has more than one hidden layer, we also call it a **deep artificial NN**.

Adding additional hidden layers

We can add any number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in an NN as additional hyperparameters that we want to optimize for a given problem task using cross-validation technique, which we discussed in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

However, the error gradients, which we will calculate later via backpropagation, will become increasingly small as more layers are added to a network. This vanishing gradient problem makes the model learning more challenging. Therefore, special algorithms have been developed to help train such DNN structures; this is known as **deep learning**.

As shown in the preceding figure, we denote the *i*th activation unit in the *l*th layer as $a_i^{(l)}$. To make the math and code implementations a bit more intuitive, we will not use numerical indices to refer to layers, but we will use the *in* superscript for the input layer, the *h* superscript for the hidden layer, and the *out* superscript for the output layer. For instance, $a_i^{(in)}$ refers to the *i*th value in the input layer, $a_i^{(h)}$ refers to the *i*th unit in the hidden layer, and $a_i^{(out)}$ refers to the *i*th unit in the output layer. Here, the activation units $a_0^{(in)}$ and $a_0^{(h)}$ are the bias units, which we set equal to 1. The activation of the units in the input layer is just its input plus the bias unit:

$$\boldsymbol{a}^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

Notational convention for the bias units

Later in this chapter, we will implement an MLP using separate vectors for the bias units, which makes code implementation more efficient and easier to read. This concept is also used by TensorFlow, a deep learning library that we will cover in *Chapter 13, Parallelizing Neural Network Training with TensorFlow*. However, the mathematical equations that will follow would appear more complex or convoluted if we had to work with additional variables for the bias. Note that the computation via appending 1s to the input vector (as shown previously) and using a weight variable as bias is exactly the same as operating with separate bias vectors; it is merely a different convention.

Each unit in layer *l* is connected to all units in layer *l* + 1 via a weight coefficient. For example, the connection between the *k*th unit in layer *l* to the *j*th unit in layer *l* + 1 will be written as $w_{k,j}^{(l)}$. Referring back to the previous figure, we denote the weight matrix that connects the input to the hidden layer as $\boldsymbol{W}^{(h)}$, and we write the matrix that connects the hidden layer to the output layer as $\boldsymbol{W}^{(out)}$.

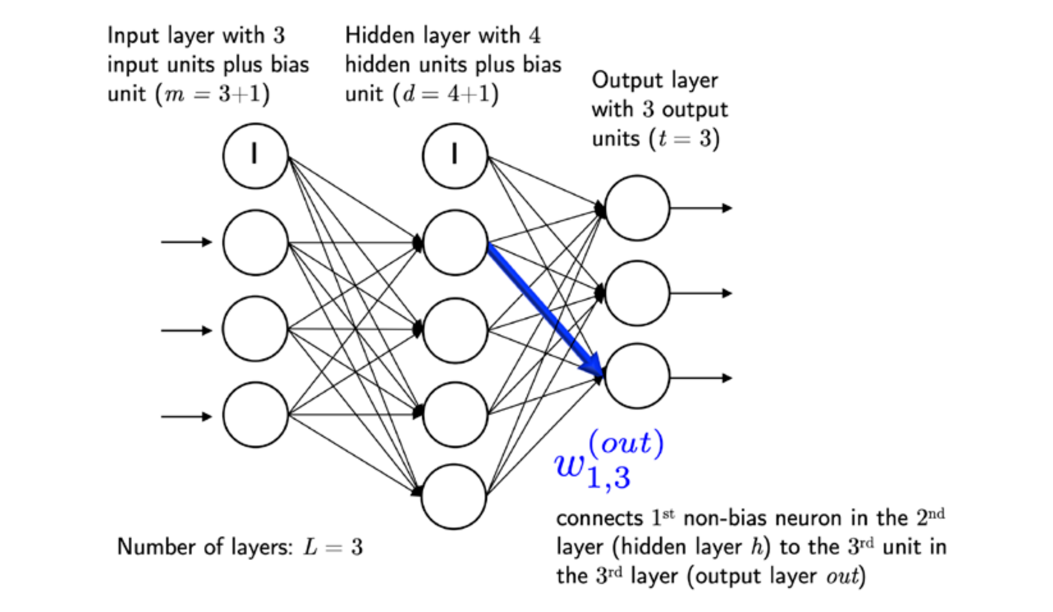
While one unit in the output layer would suffice for a binary classification task, we saw a more general form of NN in the preceding figure, which allows us to perform multiclass classification via a generalization of the **one-versus-all** (**OvA**) technique. To better understand how this works, remember the **one-hot** representation of categorical variables that we introduced in *Chapter 4, Building Good Training Datasets – Data Preprocessing*.

For example, we can encode the three class labels in the familiar Iris dataset (*0=Setosa, 1=Versicolor, 2=Virginica*) as follows:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training dataset.

If you are new to NN representations, the indexing notation (subscripts and superscripts) may look a little bit confusing at first. What may seem overly complicated at first will make much more sense in later sections when we vectorize the NN representation. As introduced earlier, we summarize the weights that connect the input and hidden layers by a matrix $\boldsymbol{W}^{(h)} \in \mathbb{R}^{m\times d}$, where d is the number of hidden units and m is the number of input units including the bias unit. Since it is important to internalize this notation to follow the concepts later in this chapter, let's summarize what we have just learned in a descriptive illustration of a simplified 3-4-3 MLP:


Input layer with 3 input units plus bias unit ($m = 3+1$)
Hidden layer with 4 hidden units plus bias unit ($d = 4+1$)
Output layer with 3 output units ($t = 3$)
I
I
$w_{1,3}^{(out)}$
connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer out)
Number of layers: $L = 3$

Activating a neural network via forward propagation

In this section, we will describe the process of **forward propagation** to calculate the output of an MLP model. To understand how it fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the error that we want to minimize using a cost function that we will describe later.
3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

Finally, after we repeat these three steps for multiple epochs and learn the weights of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

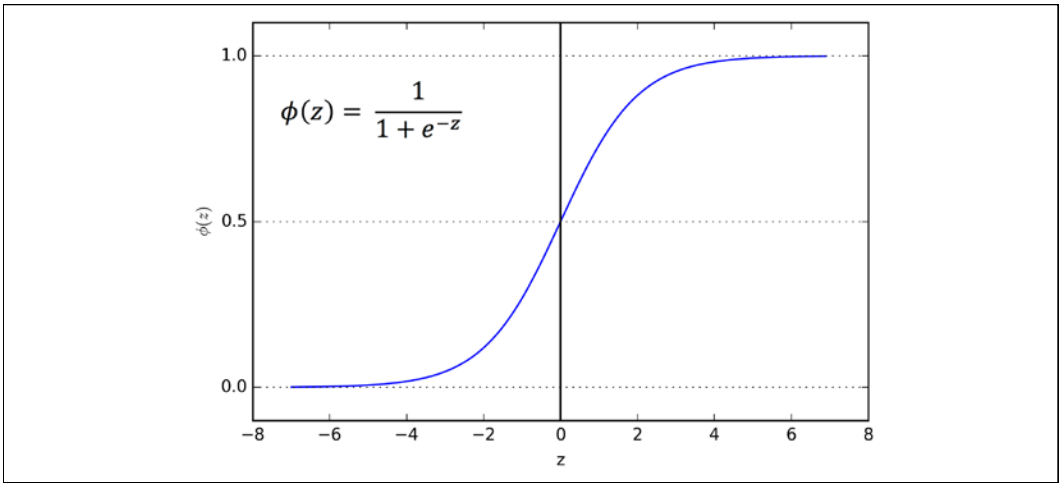
Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation unit of the hidden layer $a_1^{(h)}$ as follows:

$$\begin{aligned} z_1^{(h)} &= a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \cdots + a_m^{(in)} w_{m,1}^{(h)} \\ a_1^{(h)} &= \phi\left(z_1^{(h)}\right) \end{aligned}$$

Here, $z_1^{(h)}$ is the net input and $\phi(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the sigmoid (logistic) activation function that we remember from the section about logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

As you may recall, the sigmoid function is an *S*-shaped curve that maps the net input *z* onto a logistic distribution in the range 0 to 1, which cuts the *y*-axis at *z* = 0, as shown in the following graph:



MLP is a typical example of a feedforward artificial NN. The term **feedforward** refers to the fact that each layer serves as the input to the next layer without loops, in contrast to recurrent NNs—an architecture that we will discuss later in this chapter and discuss in more detail in *Chapter 16, Modeling Sequential Data Using Recurrent Neural Networks*. The term *multilayer perceptron* may sound a little bit confusing since the artificial neurons in this network architecture are typically sigmoid units, not perceptrons. We can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1.

For purposes of code efficiency and readability, we will now write the activation in a more compact form using the concepts of basic linear algebra, which will allow us to vectorize our code implementation via NumPy rather than writing multiple nested and computationally expensive Python `for` loops:

$$\boldsymbol{z}^{(h)} = \boldsymbol{a}^{(in)}\boldsymbol{W}^{(h)}$$

$$\boldsymbol{a}^{(h)} = \phi\left(\boldsymbol{z}^{(h)}\right)$$

Here, $\boldsymbol{a}^{(in)}$ is our $1 \times m$ dimensional feature vector of a sample $\boldsymbol{x}^{(in)}$ plus a bias unit.

$\boldsymbol{W}^{(h)}$ is an $m \times d$ dimensional weight matrix where d is the number of units in the hidden layer. After matrix-vector multiplication, we obtain the $1 \times d$ dimensional net input vector $\boldsymbol{z}^{(h)}$ to calculate the activation $\boldsymbol{a}^{(h)}$ (where $\boldsymbol{a}^{(h)} \epsilon \mathbb{R}^{1\times d}$).

Furthermore, we can generalize this computation to all n examples in the training dataset:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{A}^{(in)}\boldsymbol{W}^{(h)}$$

Here, $\boldsymbol{A}^{(in)}$ is now an $n \times m$ matrix, and the matrix-matrix multiplication will result in an $n \times d$ dimensional net input matrix $\boldsymbol{Z}^{(h)}$. Finally, we apply the activation function $\phi(\cdot)$ to each value in the net input matrix to get the $n \times d$ activation matrix the next layer (here, the output layer):

$$\boldsymbol{A}^{(h)} = \phi\left(\boldsymbol{Z}^{(h)}\right)$$

Similarly, we can write the activation of the output layer in vectorized form for multiple examples:

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)}$$

Here, we multiply the $d \times t$ matrix $\boldsymbol{W}^{(out)}$ (t is the number of output units) by the $n \times d$ dimensional matrix $\boldsymbol{A}^{(h)}$ to obtain the $n \times t$ dimensional matrix $\boldsymbol{Z}^{(out)}$ (the columns in this matrix represent the outputs for each sample).

Lastly, we apply the sigmoid activation function to obtain the continuous valued output of our network:

$$\boldsymbol{A}^{(out)} = \phi\left(\boldsymbol{Z}^{(out)}\right), \qquad \boldsymbol{A}^{(out)} \in \mathbb{R}^{n \times t}$$

Classifying handwritten digits

In the previous section, we covered a lot of the theory around NNs, which can be a little bit overwhelming if you are new to this topic. Before we continue with the discussion of the algorithm for learning the weights of the MLP model, backpropagation, let's take a short break from the theory and see an NN in action.

Additional resources on backpropagation

The NN theory can be quite complex; thus, it is recommended that you refer to two additional resources, which cover some of the concepts that we discuss in this chapter in more detail:

- *Chapter 6*, *Deep Feedforward Networks*, *Deep Learning*, *I. Goodfellow*, *Y. Bengio*, and *A. Courville*, MIT Press, *2016* (Manuscripts freely accessible at `http://www.deeplearningbook.org`).

- *Pattern Recognition and Machine Learning*, *C. M. Bishop* and others, Volume 1. *Springer New York*, *2006*.
- Lecture slides from the deep learning course at the University of Wisconsin–Madison:
    1. `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L08_logistic_slides.pdf`
    2. `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L09_mlp_slides.pdf`

In this section, we will implement and train our first multilayer NN to classify handwritten digits from the popular **Mixed National Institute of Standards and Technology** (**MNIST**) dataset that has been constructed by Yann LeCun and others and serves as a popular benchmark dataset for machine learning algorithms (*Gradient-Based Learning Applied to Document Recognition*, *Y. LeCun*, *L. Bottou*, *Y. Bengio*, and *P. Haffner*, *Proceedings of the IEEE*, 86(11): 2278-2324, *November 1998*).

Obtaining and preparing the MNIST dataset

The MNIST dataset is publicly available at `http://yann.lecun.com/exdb/mnist/` and consists of the following four parts:

- **Training dataset images**: `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB unzipped, and 60,000 examples)
- **Training dataset labels**: `train-labels-idx1-ubyte.gz` (29 KB, 60 KB unzipped, and 60,000 labels)
- **Test dataset images**: `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB unzipped, and 10,000 examples)
- **Test dataset labels**: `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB unzipped, and 10,000 labels)

The MNIST dataset was constructed from two datasets of the US **National Institute of Standards and Technology** (**NIST**). The training dataset consists of handwritten digits from 250 different people, 50 percent high school students and 50 percent employees from the Census Bureau. Note that the test dataset contains handwritten digits from different people following the same split.

After you download the files, it is recommended that you unzip them using the Unix/Linux `gzip` tool from the terminal for efficiency, using the following command in your local MNIST download directory:

```
gzip *ubyte.gz -d
```

Alternatively, you can use your favorite unzipping tool if you are working with a machine running Microsoft Windows.

The images are stored in byte format, and we will read them into NumPy arrays that we will use to train and test our MLP implementation. In order to do that, we will define the following helper function:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from 'path'"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(
                             len(labels), 784)
        images = ((images / 255.) - .5) * 2

    return images, labels
```

The `load_mnist` function returns two arrays, the first being an $n \times m$ dimensional NumPy array (`images`), where n is the number of examples and m is the number of features (here, pixels). The training dataset consists of 60,000 training digits and the test dataset contains 10,000 examples, respectively.

The images in the MNIST dataset consist of 28 × 28 pixels, and each pixel is represented by a grayscale intensity value. Here, we unroll the 28 × 28 pixels into one-dimensional row vectors, which represent the rows in our `images` array (784 per row or image). The second array (`labels`) returned by the `load_mnist` function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits.

The way we read in the image might seem a little bit strange at first:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.uint8)
```

To understand how those two lines of code work, let's take a look at the dataset description from the MNIST website:

```
[offset] [type]           [value]          [description]
0000      32 bit integer  0x00000801(2049) magic number (MSB first)
0004      32 bit integer  60000            number of items
0008      unsigned byte   ??               label
0009      unsigned byte   ??               label
........
xxxx      unsigned byte   ??               label
```

Using the two preceding lines of code, we first read in the magic number, which is a description of the file protocol, as well as the number of items (`n`) from the file buffer, before we load the following bytes into a NumPy array using the `fromfile` method. The `fmt` parameter value, `'>II'`, that we passed as an argument to `struct.unpack` can be composed into the two following parts:

- `>`: This is big-endian—it defines the order in which a sequence of bytes is stored; if you are unfamiliar with the terms big-endian and little-endian, you can find an excellent article about *Endianness* on Wikipedia: `https://en.wikipedia.org/wiki/Endianness`
- `I`: This is an unsigned integer

Finally, we also normalized the pixels values in MNIST to the range –1 to 1 (originally 0 to 255) via the following code line:

```
images = ((images / 255.) - .5) * 2
```

The reason behind this is that gradient-based optimization is much more stable under these conditions as discussed in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*. Note that we scaled the images on a pixel-by-pixel basis, which is different from the feature scaling approach that we took in previous chapters.

Previously, we derived scaling parameters from the training dataset and used these to scale each column in the training dataset and test dataset. However, when working with image pixels, centering them at zero and rescaling them to a [–1, 1] range is also common and usually works well in practice.

Batch normalization

A commonly used trick for improving convergence in gradient-based optimization through input scaling is *batch normalization*, which is an advanced topic that we will cover in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*. Also, you can read more about batch normalization in the excellent research article *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* by *Sergey Ioffe* and *Christian Szegedy* (2015, `https://arxiv.org/abs/1502.03167`).

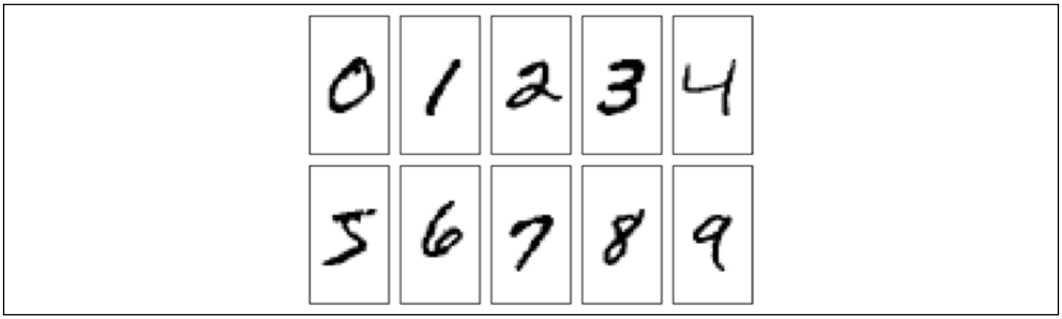
By executing the following code, we will now load the 60,000 training instances as well as the 10,000 test examples from the local directory where we unzipped the MNIST dataset. (In the following code snippet, it is assumed that the downloaded MNIST files were unzipped to the same directory in which this code was executed.)

```
>>> X_train, y_train = load_mnist('', kind='train')
>>> print('Rows: %d, columns: %d'
...       % (X_train.shape[0], X_train.shape[1]))
Rows: 60000, columns: 784

>>> X_test, y_test = load_mnist('', kind='t10k')
>>> print('Rows: %d, columns: %d'
...       % (X_test.shape[0], X_test.shape[1]))
Rows: 10000, columns: 784
```

To get an idea of how those images in MNIST look, let's visualize examples of the digits 0-9 after reshaping the 784-pixel vectors from our feature matrix into the original 28 × 28 image that we can plot via Matplotlib's `imshow` function:

```
>>> import matplotlib.pyplot as plt

>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                        sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')

>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

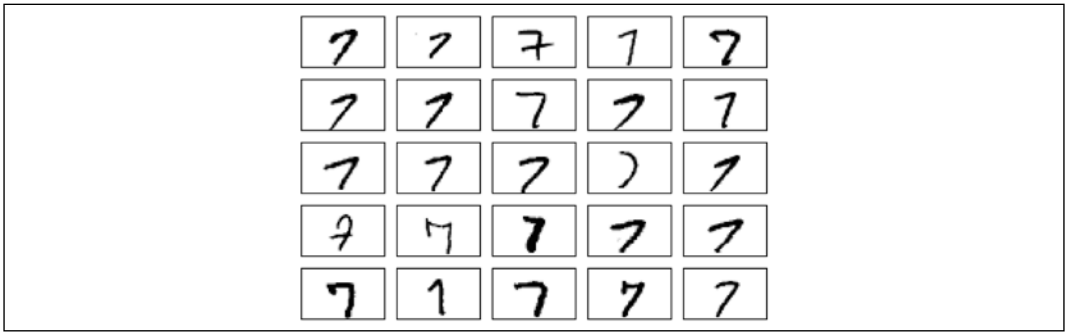
We should now see a plot of the 2 × 5 subfigures showing a representative image of each unique digit:



In addition, let's also plot multiple examples of the same digit to see how different the handwriting for each really is:

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should now see the first 25 variants of the digit 7:



After we've gone through all the previous steps, it is a good idea to save the scaled images in a format that we can load more quickly into a new Python session to avoid the overhead of reading in and processing the data again. When we are working with NumPy arrays, an efficient and convenient method to save multidimensional arrays to disk is NumPy's `savez` function. (The official documentation can be found here: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html`.)

In short, the `savez` function is analogous to Python's `pickle` module, which we used in *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, but optimized for storing NumPy arrays. The `savez` function creates zipped archives of our data, producing `.npz` files that contain files in the `.npy` format; if you want to learn more about this format, you can find a nice explanation, including a discussion about advantages and disadvantages, in the NumPy documentation: `https://docs.scipy.org/doc/numpy/neps/npy-format.html`. Furthermore, instead of using `savez`, we will use `savez_compressed`, which uses the same syntax as `savez`, but further compresses the output file down to substantially smaller file sizes (approximately 22 MB versus approximately 400 MB in this case). The following code snippet will save both the training and test datasets to the archive file `mnist_scaled.npz`:

```
>>> import numpy as np

>>> np.savez_compressed('mnist_scaled.npz',
...                     X_train=X_train,
...                     y_train=y_train,
...                     X_test=X_test,
...                     y_test=y_test)
```

After we create the `.npz` files, we can load the preprocessed MNIST image arrays using NumPy's `load` function as follows:

```
>>> mnist = np.load('mnist_scaled.npz')
```

The `mnist` variable now refers to an object that can access the four data arrays that we provided as keyword arguments to the `savez_compressed` function. These input arrays are now listed under the `files` attribute list of the `mnist` object:

```
>>> mnist.files
['X_train', 'y_train', 'X_test', 'y_test']
```

For instance, to load the training data into our current Python session, we will access the `X_train` array as follows (similar to a Python dictionary):

```
>>> X_train = mnist['X_train']
```

Using a list comprehension, we can retrieve all four data arrays as follows:

```
>>> X_train, y_train, X_test, y_test = [mnist[f] for
...                                     f in mnist.files]
```

Note that while the preceding `np.savez_compressed` and `np.load` examples are not essential for executing the code in this chapter, they serve as a demonstration of how to save and load NumPy arrays conveniently and efficiently.

Loading MNIST using scikit-learn

Using scikit-learn's new `fetch_openml` function, it is now also possible to load the MNIST dataset more conveniently. For example, you can use the following code to create a 50,000-example training dataset and and a 10,000-example test dataset by fetching the dataset from `https://www.openml.org/d/554`:

```
>>> from sklearn.datasets import fetch_openml
>>> from sklearn.model_selection import train_test_
split
>>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>>> y = y.astype(int)
>>> X = ((X / 255.) - .5) * 2
>>> X_train, X_test, y_train, y_test =\
...  train_test_split(
...     X, y, test_size=10000,
...     random_state=123, stratify=y)
```

Please note that the distribution of MNIST records into training and test datasets will be different from the manual approach outlined in this section. Thus, you will observe slightly different results in the following sections if you load the dataset using the `fetch_openml` and `train_test_split` functions.

Implementing a multilayer perceptron

In this subsection, we will now implement an MLP from scratch to classify the images in the MNIST dataset. To keep things simple, we will implement an MLP with only one hidden layer. Since the approach may seem a little bit complicated at first, you are encouraged to download the sample code for this chapter from the Packt Publishing website or from GitHub (`https://github.com/rasbt/python-machine-learning-book-3rd-edition`) so that you can view this MLP implementation annotated with comments and syntax highlighting for better readability.

If you are not running the code from the accompanying Jupyter Notebook file or don't have access to the Internet, copy the `NeuralNetMLP` code from this chapter into a Python script file in your current working directory (for example, `neuralnet.py`), which you can then import into your current Python session via the following command:

```
from neuralnet import NeuralNetMLP
```

The code will contain parts that we have not talked about yet, such as the backpropagation algorithm, but most of the code should look familiar to you based on the Adaline implementation in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, and the discussion of forward propagation in earlier sections.

Do not worry if not all of the code makes immediate sense to you; we will follow up on certain parts later in this chapter. However, going over the code at this stage can make it easier to follow the theory later.

The following is the implementation of an MLP:

```
import numpy as np
import sys

class NeuralNetMLP(object):
    """ Feedforward neural network / Multi-layer perceptron
classifier.


    Parameters
    ------------
    n_hidden : int (default: 30)
        Number of hidden units.
    l2 : float (default: 0.)
        Lambda value for L2-regularization.
        No regularization if l2=0. (default)
    epochs : int (default: 100)
        Number of passes over the training set.
    eta : float (default: 0.001)
        Learning rate.
    shuffle : bool (default: True)
        Shuffles training data every epoch
        if True to prevent circles.
    minibatch_size : int (default: 1)
        Number of training examples per minibatch.
    seed : int (default: None)
```

```
            Random seed for initializing weights and shuffling.

        Attributes
        -----------
        eval_ : dict
            Dictionary collecting the cost, training accuracy,
            and validation accuracy for each epoch during training.

        """
        def __init__(self, n_hidden=30,
                     l2=0., epochs=100, eta=0.001,
                     shuffle=True, minibatch_size=1, seed=None):

            self.random = np.random.RandomState(seed)
            self.n_hidden = n_hidden
            self.l2 = l2
            self.epochs = epochs
            self.eta = eta
            self.shuffle = shuffle
            self.minibatch_size = minibatch_size

        def _onehot(self, y, n_classes):
            """Encode labels into one-hot representation

            Parameters
            ------------
            y : array, shape = [n_examples]
                Target values.

            Returns
            -----------
            onehot : array, shape = (n_examples, n_labels)

            """
            onehot = np.zeros((n_classes, y.shape[0]))
            for idx, val in enumerate(y.astype(int)):
                onehot[val, idx] = 1.
            return onehot.T

        def _sigmoid(self, z):
            """Compute logistic function (sigmoid)"""
            return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

        def _forward(self, X):
```

```
        """Compute forward propagation step"""

        # step 1: net input of hidden layer
        # [n_examples, n_features] dot [n_features, n_hidden]
        # -> [n_examples, n_hidden]
        z_h = np.dot(X, self.w_h) + self.b_h

        # step 2: activation of hidden layer
        a_h = self._sigmoid(z_h)

        # step 3: net input of output layer
        # [n_examples, n_hidden] dot [n_hidden, n_classlabels]
        # -> [n_examples, n_classlabels]

        z_out = np.dot(a_h, self.w_out) + self.b_out
        # step 4: activation output layer
        a_out = self._sigmoid(z_out)

        return z_h, a_h, z_out, a_out

    def _compute_cost(self, y_enc, output):
        """Compute cost function.

        Parameters
        ----------
        y_enc : array, shape = (n_examples, n_labels)
            one-hot encoded class labels.
        output : array, shape = [n_examples, n_output_units]
            Activation of the output layer (forward propagation)

        Returns
        ---------
        cost : float
            Regularized cost

        """
        L2_term = (self.l2 *
                   (np.sum(self.w_h ** 2.) +
                    np.sum(self.w_out ** 2.)))

        term1 = -y_enc * (np.log(output))
        term2 = (1. - y_enc) * np.log(1. - output)
        cost = np.sum(term1 - term2) + L2_term
        return cost
    def predict(self, X):
        """Predict class labels

        Parameters
        -----------
        X : array, shape = [n_examples, n_features]
            Input layer with original features.

        Returns:
        ----------
        y_pred : array, shape = [n_examples]
            Predicted class labels.

        """
        z_h, a_h, z_out, a_out = self._forward(X)
        y_pred = np.argmax(z_out, axis=1)
        return y_pred

    def fit(self, X_train, y_train, X_valid, y_valid):
        """ Learn weights from training data.

        Parameters
        -----------
        X_train : array, shape = [n_examples, n_features]
            Input layer with original features.
        y_train : array, shape = [n_examples]
            Target class labels.
        X_valid : array, shape = [n_examples, n_features]
            Sample features for validation during training
        y_valid : array, shape = [n_examples]
            Sample labels for validation during training

        Returns:
        ----------
        self

        """
        n_output = np.unique(y_train).shape[0] # no. of class
                                               #labels
        n_features = X_train.shape[1]

        ########################
        # Weight initialization
        ########################

        # weights for input -> hidden
        self.b_h = np.zeros(self.n_hidden)
        self.w_h = self.random.normal(loc=0.0, scale=0.1,
                                      size=(n_features,
                                            self.n_hidden))

        # weights for hidden -> output
        self.b_out = np.zeros(n_output)
        self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                        size=(self.n_hidden,
                                              n_output))

        epoch_strlen = len(str(self.epochs)) # for progr. format.
        self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': \
                      []}

        y_train_enc = self._onehot(y_train, n_output)

        # iterate over training epochs
        for i in range(self.epochs):

            # iterate over minibatches
            indices = np.arange(X_train.shape[0])

            if self.shuffle:
                self.random.shuffle(indices)

            for start_idx in range(0, indices.shape[0] -\
                                   self.minibatch_size +\
                                   1, self.minibatch_size):
                batch_idx = indices[start_idx:start_idx +\
                                    self.minibatch_size]

                # forward propagation
                z_h, a_h, z_out, a_out = \
                    self._forward(X_train[batch_idx])

                ##################
                # Backpropagation
                ##################

                # [n_examples, n_classlabels]
                delta_out = a_out - y_train_enc[batch_idx]
                # [n_examples, n_hidden]
                sigmoid_derivative_h = a_h * (1. - a_h)

                # [n_examples, n_classlabels] dot [n_classlabels,
                #                                  n_hidden]
                # -> [n_examples, n_hidden]
                delta_h = (np.dot(delta_out, self.w_out.T) *
                           sigmoid_derivative_h)

                # [n_features, n_examples] dot [n_examples,
                #                                n_hidden]
                # -> [n_features, n_hidden]
                grad_w_h = np.dot(X_train[batch_idx].T, delta_h)
                grad_b_h = np.sum(delta_h, axis=0)

                # [n_hidden, n_examples] dot [n_examples,
                #                             n_classlabels]
                # -> [n_hidden, n_classlabels]
                grad_w_out = np.dot(a_h.T, delta_out)
                grad_b_out = np.sum(delta_out, axis=0)

                # Regularization and weight updates
                delta_w_h = (grad_w_h + self.l2*self.w_h)
                delta_b_h = grad_b_h # bias is not regularized
                self.w_h -= self.eta * delta_w_h
                self.b_h -= self.eta * delta_b_h

                delta_w_out = (grad_w_out + self.l2*self.w_out)
                delta_b_out = grad_b_out # bias is not regularized
                self.w_out -= self.eta * delta_w_out
                self.b_out -= self.eta * delta_b_out

            #############
            # Evaluation
            #############

            # Evaluation after each epoch during training
            z_h, a_h, z_out, a_out = self._forward(X_train)

            cost = self._compute_cost(y_enc=y_train_enc,
                                      output=a_out)

            y_train_pred = self.predict(X_train)
            y_valid_pred = self.predict(X_valid)
            train_acc = ((np.sum(y_train ==
                          y_train_pred)).astype(np.float) /
                         X_train.shape[0])
            valid_acc = ((np.sum(y_valid ==
                          y_valid_pred)).astype(np.float) /
                         X_valid.shape[0])

            sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                             '| Train/Valid Acc.: %.2f%%/%.2f%% '
                              %
                             (epoch_strlen, i+1, self.epochs,
                              cost,
                              train_acc*100, valid_acc*100))
            sys.stderr.flush()

            self.eval_['cost'].append(cost)
            self.eval_['train_acc'].append(train_acc)
            self.eval_['valid_acc'].append(valid_acc)

        return self
```

After executing this code, we next initialize a new 784-100-10 MLP—an NN with 784 input units (`n_features`), 100 hidden units (`n_hidden`), and 10 output units (`n_output`):

```
>>> nn = NeuralNetMLP(n_hidden=100,
...                   l2=0.01,
...                   epochs=200,
...                   eta=0.0005,
...                   minibatch_size=100,
...                   shuffle=True,
...                   seed=1)
```

If you read through the `NeuralNetMLP` code, you've probably already guessed what these parameters are for. Here, you find a short summary of them:

- `l2`: This is the λ parameter for L2 regularization to decrease the degree of overfitting.
- `epochs`: This is the number of passes over the training dataset.
- `eta`: This is the learning rate η.
- `shuffle`: This is for shuffling the training set prior to every epoch to prevent the algorithm getting stuck in circles.
- `seed`: This is a random seed for shuffling and weight initialization.

- `minibatch_size`: This is the number of training examples in each mini-batch when splitting the training data in each epoch for SGD. The gradient is computed for each mini-batch separately instead of the entire training data for faster learning.

Next, we train the MLP using 55,000 examples from the already shuffled MNIST training dataset and use the remaining 5,000 examples for validation during training. Note that training the NN may take up to five minutes on standard desktop computer hardware.

As you may have noticed from the preceding code, we implemented the `fit` method so that it takes four input arguments: training images, training labels, validation images, and validation labels. In NN training, it is really useful to compare training and validation accuracy, which helps us judge whether the network model performs well, given the architecture and hyperparameters. For example, if we observe a low training and validation accuracy, there is likely an issue with the training dataset, or the hyperparameters settings are not ideal. A relatively large gap between the training and the validation accuracy indicated that the model is likely overfitting the training dataset so that we want to reduce the number of parameters in the model or increase the regularization strength. If both the training and validation accuracies are high, the model is likely to generalize well to new data, for example, the test dataset, which we use for the final model evaluation.

In general, training (deep) NNs is relatively expensive compared with the other models we've discussed so far. Thus, we want to stop it early in certain circumstances and start over with different hyperparameter settings. On the other hand, if we find that it increasingly tends to overfit the training data (noticeable by an increasing gap between training and validation dataset performance), we may want to stop the training early as well.

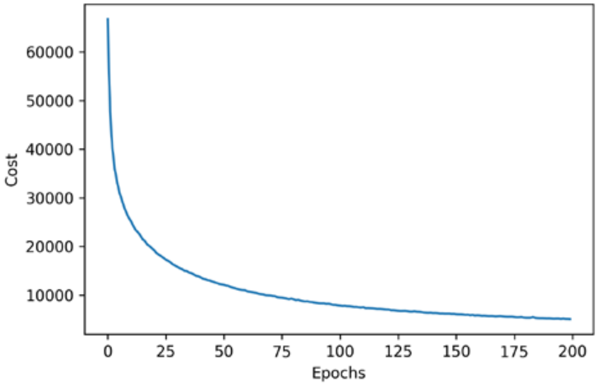
Now, to start the training, we execute the following code:

```
>>> nn.fit(X_train=X_train[:55000],
...        y_train=y_train[:55000],
...        X_valid=X_train[55000:],
...        y_valid=y_train[55000:])
200/200 | Cost: 5065.78 | Train/Valid Acc.: 99.28%/97.98%
```

In our `NeuralNetMLP` implementation, we also defined an `eval_` attribute that collects the cost, training, and validation accuracy for each epoch so that we can visualize the results using Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(nn.epochs), nn.eval_['cost'])
>>> plt.ylabel('Cost')
>>> plt.xlabel('Epochs')
>>> plt.show()
```

The preceding code plots the cost over the 200 epochs, as shown in the following graph:

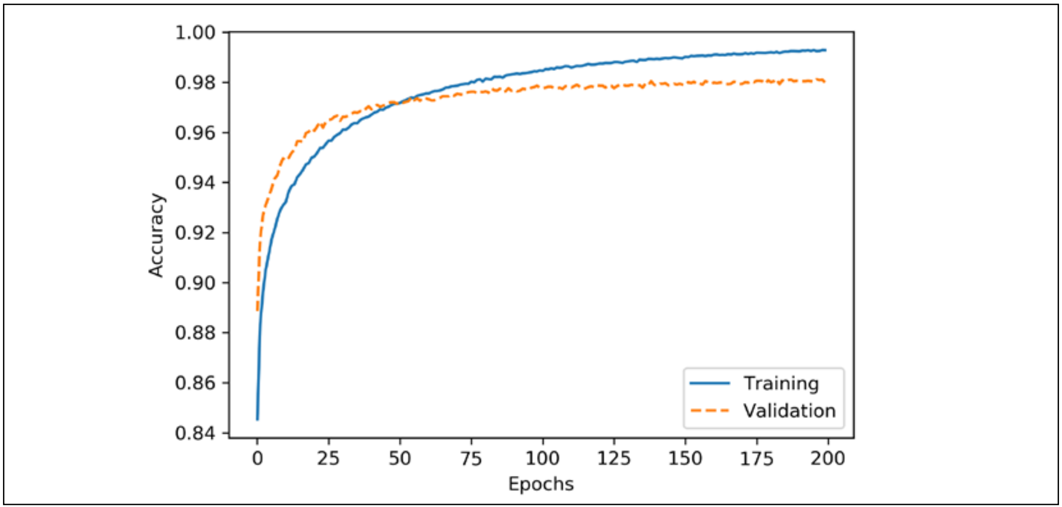


As we can see, the cost decreased substantially during the first 100 epochs and seems to slowly converge in the last 100 epochs. However, the small slope between epoch `175` and epoch `200` indicates that the cost would further decrease with a training over additional epochs.

Next, let's take a look at the training and validation accuracy:

```
>>> plt.plot(range(nn.epochs), nn.eval_['train_acc'],
...          label='training')
>>> plt.plot(range(nn.epochs), nn.eval_['valid_acc'],
...          label='validation', linestyle='--')
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Epochs')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

The preceding code examples plot those accuracy values over the 200 training epochs, as shown in the following figure:



The plot reveals that the gap between training and validation accuracy increases as we train for more epochs. At approximately the 50th epoch, the training and validation accuracy values are equal, and then, the network starts overfitting the training data.

Note that this example was chosen deliberately to illustrate the effect of overfitting and demonstrate why it is useful to compare the validation and training accuracy values during training. One way to decrease the effect of overfitting is to increase the regularization strength—for example, by setting `l2=0.1`. Another useful technique to tackle overfitting in NNs is *dropout*, which will be covered in *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*.

Finally, let's evaluate the generalization performance of the model by calculating the prediction accuracy on the test dataset:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = (np.sum(y_test == y_test_pred)
...         .astype(np.float) / X_test.shape[0])
>>> print('Test accuracy: %.2f%%' % (acc * 100))
Test accuracy: 97.54%
```

Despite the slight overfitting on the training data, our relatively simple one-hidden-layer NN achieved a relatively good performance on the test dataset, similar to the validation dataset accuracy (97.98 percent).

To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, and the learning rate, or use various other tricks that have been developed over the years but are beyond the scope of this book. In *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*, you will learn about a different NN architecture that is known for its good performance on image datasets. Also, the chapter will introduce additional performance-enhancing tricks such as adaptive learning rates, more sophisticated SGD-based optimization algorithms, batch normalization, and dropout.

Other common tricks that are beyond the scope of the following chapters include:

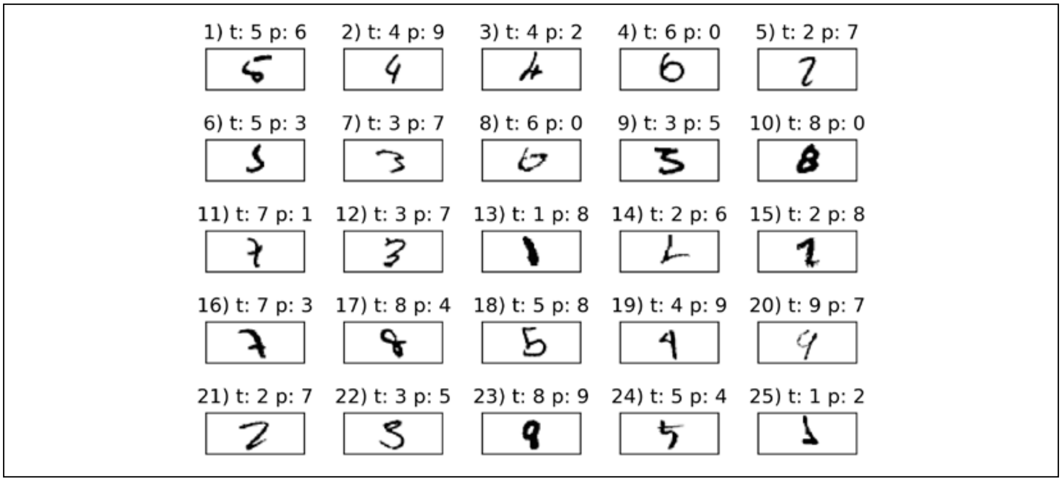
- Adding skip-connections, which are the main contribution of residual NNs (*Deep residual learning for image recognition*. K. He, X. Zhang, S. Ren, J. Sun (2016). In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770-778)
- Using learning rate schedulers that change the learning rate during training (*Cyclical learning rates for training neural networks*. L.N. Smith (2017). In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464-472)
- Attaching loss functions to earlier layers in the networks as it's being done in the popular Inception v3 architecture (*Rethinking the Inception architecture for computer vision*. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna (2016). In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826)

Lastly, let's take a look at some of the images that our MLP struggles with:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab = y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d) t: %d p: %d'
...     % (i+1, correct_lab[i], miscl_lab[i]))
```

```
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a 5 × 5 subplot matrix where the first number in the subtitles indicates the plot index, the second number represents the true class label (`t`), and the third number stands for the predicted class label (`p`):



As we can see in the preceding figure, some of those images are even challenging for us humans to classify correctly. For example, the 6 in subplot `8` really looks like a carelessly drawn 0, and the 8 in subplot `23` could be a 9 due to the narrow lower part combined with the bold line.

Training an artificial neural network

Now that we have seen an NN in action and have gained a basic understanding of how it works by looking over the code, let's dig a little bit deeper into some of the concepts, such as the logistic cost function and the backpropagation algorithm that we implemented to learn the weights.

Computing the logistic cost function

The logistic cost function that we implemented as the `_compute_cost` method is actually pretty simple to follow since it is the same cost function that we described in the logistic regression section in *Chapter 3, A Tour of Machine Learning Classifiers Using scikit-learn*:

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{[i]} \log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) \log\left(1 - a^{[i]}\right)$$

Here, $a^{[i]}$ is the sigmoid activation of the ith sample in the dataset, which we compute in the forward propagation step:

$$a^{[i]} = \phi\left(z^{[i]}\right)$$

Again, note that in this context, the superscript [i] is an index for training examples, not layers.

Now, let's add a regularization term, which allows us to reduce the degree of overfitting. As you recall from earlier chapters, the L2 regularization term is defined as follows (remember that we don't regularize the bias units):

$$L2 = \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

By adding the L2 regularization term to our logistic cost function, we obtain the following equation:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n} y^{[i]} \log\left(a^{[i]}\right) + \left(1 - y^{[i]}\right) \log\left(1 - a^{[i]}\right)\right] + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2$$

Previously, we implemented an MLP for multiclass classification that returns an output vector of t elements that we need to compare to the $t \times 1$ dimensional target vector in the one-hot encoding representation. If we predict the class label of an input image with class label 2, using this MLP, the activation of the third layer and the target may look like this:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \qquad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, we need to generalize the logistic cost function to all t activation units in our network.

The cost function (without the regularization term) becomes the following:

$$J(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]}\log\left(a_j^{[i]}\right) + \left(1 - y_j^{[i]}\right)\log\left(1 - a_j^{[i]}\right)$$

Here, again, the superscript [*i*] is the index of a particular sample in our training dataset.

The following generalized regularization term may look a little bit complicated at first, but here we are just calculating the sum of all weights of an *l* layer (without the bias term) that we added to the first column:

$$J(\boldsymbol{W}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]}\, log\left(a_j^{[i]}\right) + \left(1 - y_j^{[i]}\right) log\left(1 - a_j^{[i]}\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Here, u_l refers to the number of units in a given layer *l*, and the following expression represents the penalty term:

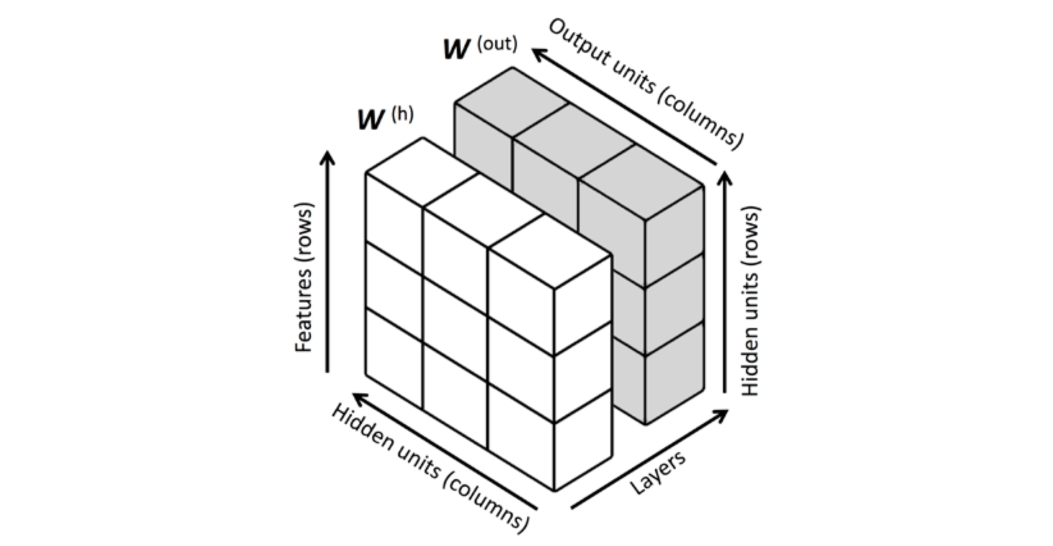
$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Remember that our goal is to minimize the cost function *J*(***W***); thus, we need to calculate the partial derivative of the parameters ***W*** with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\boldsymbol{W})$$

In the next section, we will talk about the backpropagation algorithm, which allows us to calculate those partial derivatives to minimize the cost function.

Note that ***W*** consists of multiple matrices. In an MLP with one hidden layer, we have the weight matrix, $\boldsymbol{W}^{(h)}$, which connects the input to the hidden layer, and $\boldsymbol{W}^{(out)}$, which connects the hidden layer to the output layer. A visualization of the three-dimensional tensor ***W*** is provided in the following figure:



In this simplified figure, it may seem that both $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ have the same number of rows and columns, which is typically not the case unless we initialize an MLP with the same number of hidden units, output units, and input features.

If this sounds confusing, stay tuned for the next section, where we will discuss the dimensionality of $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ in more detail in the context of the backpropagation algorithm. Also, you are encouraged to read through the code of the `NeuralNetMLP` again, which is annotated with helpful comments about the dimensionality of the different matrices and vector transformations. You can obtain the annotated code either from Packt or the book's GitHub repository at `https://github.com/rasbt/python-machine-learning-book-3rd-edition`.

Developing your understanding of backpropagation

Although backpropagation was rediscovered and popularized more than 30 years ago (*Learning representations by back-propagating errors*, *D. E. Rumelhart*, *G. E. Hinton*, and *R. J. Williams*, *Nature*, 323: 6088, pages 533–536, *1986*), it still remains one of the most widely used algorithms to train artificial NNs very efficiently. If you are interested in additional references regarding the history of backpropagation, Juergen Schmidhuber wrote a nice survey article, *Who Invented Backpropagation?*, which you can find online at `http://people.idsia.ch/~juergen/who-invented-backpropagation.html`.

This section will provide both a short, clear summary and the bigger picture of how this fascinating algorithm works before we dive into more mathematical details. In essence, we can think of backpropagation as a very computationally efficient approach to compute the partial derivatives of a complex cost function in multilayer NNs. Here, our goal is to use those derivatives to learn the weight coefficients for parameterizing such a multilayer artificial NN. The challenge in the parameterization of NNs. is that we are typically dealing with a very large number of weight coefficients in a high-dimensional feature space. In contrast to cost functions of single-layer NNs such as Adaline or logistic regression, which we have seen in previous chapters, the error surface of an NN cost function is not convex or smooth with respect to the parameters. There are many bumps in this high-dimensional cost surface (local minima) that we have to overcome in order to find the global minimum of the cost function.

You may recall the concept of the chain rule from your introductory calculus classes. The chain rule is an approach to compute the derivative of a complex, nested function, such as $f(g(x))$, as follows:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg}\cdot\frac{dg}{dx}$$

Similarly, we can use the chain rule for an arbitrarily long function composition. For example, let's assume that we have five different functions, $f(x)$, $g(x)$, $h(x)$, $u(x)$, and $v(x)$, and let F be the function composition: $F(x) = f(g(h(u(v(x)))))$. Applying the chain rule, we can compute the derivative of this function as follows:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg}\cdot\frac{dg}{dh}\cdot\frac{dh}{du}\cdot\frac{du}{dv}\cdot\frac{dv}{dx}$$

In the context of computer algebra, a set of techniques has been developed to solve such problems very efficiently, which is also known as **automatic differentiation**. If you are interested in learning more about automatic differentiation in machine learning applications, read A. G. Baydin and B. A. Pearlmutter's article *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv:1404.7456, *2014*, which is freely available on arXiv at `http://arxiv.org/pdf/1404.7456.pdf`.

Automatic differentiation comes with two modes, the forward and reverse modes; backpropagation is simply a special case of reverse-mode automatic differentiation. The key point is that applying the chain rule in the forward mode could be quite expensive since we would have to multiply large matrices for each layer (Jacobians) that we would eventually multiply by a vector to obtain the output.

The trick of reverse mode is that we start from right to left: we multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on. Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in NN training.

A basic calculus refresher

To fully understand backpropagation, we need to borrow certain concepts from differential calculus, which is outside the scope of this book. However, you can refer to a review chapter of the most fundamental concepts, which you might find useful in this context. It discusses function derivatives, partial derivatives, gradients, and the Jacobian. This text is freely accessible at `https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf`. If you are unfamiliar with calculus or need a brief refresher, consider reading this text as an additional supporting resource before reading the next section.

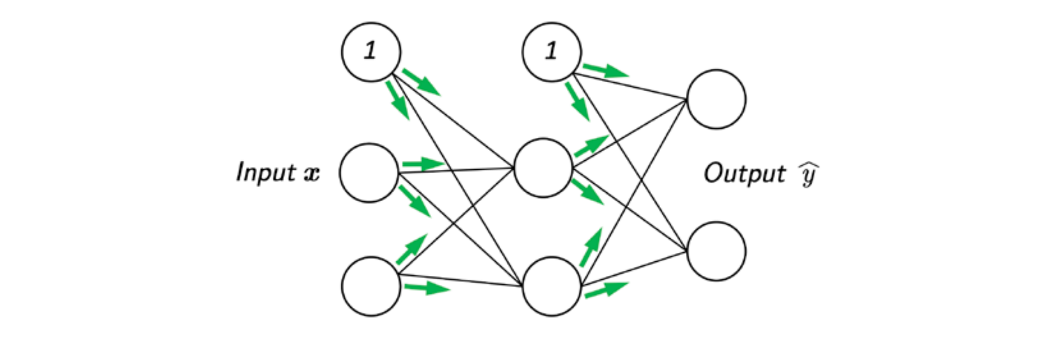
Training neural networks via backpropagation

In this section, we will go through the math of backpropagation to understand how you can learn the weights in an NN very efficiently. Depending on how comfortable you are with mathematical representations, the following equations may seem relatively complicated at first.

In a previous section, we saw how to calculate the cost as the difference between the activation of the last layer and the target class label. Now, we will see how the backpropagation algorithm works to update the weights in our MLP model from a mathematical perspective, which we implemented after the `# Backpropagation` code comment inside the `fit` method. As we recall from the beginning of this chapter, we first need to apply forward propagation in order to obtain the activation of the output layer, which we formulated as follows:

$$\begin{aligned}
\boldsymbol{Z}^{(h)} &= \boldsymbol{A}^{(in)}\boldsymbol{W}^{(h)} && \text{(net input of the hidden layer)}\\
\boldsymbol{A}^{(h)} &= \phi\left(\boldsymbol{Z}^{(h)}\right) && \text{(activation of the hidden layer)}\\
\boldsymbol{Z}^{(out)} &= \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)} && \text{(net input of the output layer)}\\
\boldsymbol{A}^{(out)} &= \phi\left(\boldsymbol{Z}^{(\boldsymbol{out})}\right) && \text{(activation of the output layer)}
\end{aligned}$$

Concisely, we just forward-propagate the input features through the connection in the network, as shown in the following illustration:



In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

$$\boldsymbol{\delta}^{(out)} = \boldsymbol{a}^{(out)} - \boldsymbol{y}$$

Here, $\boldsymbol{y}$ is the vector of the true class labels (the corresponding variable in the `NeuralNetMLP` code is `delta_out`).

Next, we calculate the error term of the hidden layer:

$$\boldsymbol{\delta}^{(h)} = \boldsymbol{\delta}^{(out)}\left(\boldsymbol{W}^{(out)}\right)^{T} \odot \frac{\partial\phi\left(z^{(h)}\right)}{\partial z^{(h)}}$$

Here, $\frac{\partial\phi\left(z^{(h)}\right)}{\partial z^{(h)}}$ is simply the derivative of the sigmoid activation function, which we computed as `sigmoid_derivative_h = a_h * (1. - a_h)` in the `fit` method of the `NeuralNetMLP`:

$$\frac{\partial\phi(z)}{\partial z} = \left(a^{(h)} \odot \left(1 - a^{(h)}\right)\right)$$

Note that the $\odot$ symbol means element-wise multiplication in this context.

Activation function derivative

Although it is not important to follow the next equations, you may be curious how the derivative of the activation function was obtained; it is summarized step by step here:

$$
\begin{aligned}
\phi'(z) &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \frac{1}{(1+e^{-z})} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \phi(z) - \big(\phi(z)\big)^2 \\
&= \phi(z)\big(1-\phi(z)\big) \\
&= a(1-\mathrm{a})
\end{aligned}
$$

Next, we compute the $\delta^{(h)}$ layer error matrix (`delta_h`) as follows:

$$\delta^{(h)} = \delta^{(out)}\left(\boldsymbol{W}^{(out)}\right)^T \odot \left(a^{(h)} \odot \left(1 - a^{(h)}\right)\right)$$

To better understand how we computed this $\delta^{(h)}$ term, let's walk through it in more detail. In the preceding equation, we used the transpose $\left(\boldsymbol{W}^{(out)}\right)^T$ of the $h \times t$-dimensional matrix $\boldsymbol{W}^{(out)}$. Here, t is the number of output class labels and h is the number of hidden units. The matrix multiplication between the $n \times t$-dimensional $\boldsymbol{\delta}^{(out)}$ matrix and the $t \times h$-dimensional matrix $\left(\boldsymbol{W}^{(out)}\right)^T$ results in an $n \times h$-dimensional matrix that we multiplied element-wise by the sigmoid derivative of the same dimension to obtain the $n \times h$-dimensional matrix $\delta^{(h)}$.

Eventually, after obtaining the δ terms, we can now write the derivation of the cost function as follows:

$$\begin{aligned}\frac{\partial}{\partial w_{i,j}^{(out)}} J(\boldsymbol{W}) &= a_j^{(h)} \delta_i^{(out)} \\ \frac{\partial}{\partial w_{i,j}^{(h)}} J(\boldsymbol{W}) &= a_j^{(in)} \delta_i^{(h)}\end{aligned}$$

Next, we need to accumulate the partial derivative of every node in each layer and the error of the node in the next layer. However, remember that we need to compute $\Delta_{i,j}^{(l)}$ for every sample in the training dataset. Thus, it is easier to implement it as a vectorized version like in our `NeuralNetMLP` code implementation:

$$\begin{aligned}\Delta^{(h)} &= \left(\boldsymbol{A}^{(in)}\right)^T \delta^{(h)} \\ \Delta^{(out)} &= \left(\boldsymbol{A}^{(h)}\right)^T \delta^{(out)}\end{aligned}$$

And after we have accumulated the partial derivatives, we can add the following regularization term:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \boldsymbol{W}^{(l)}$$

(Please note that the bias units are usually not regularized.)

The two previous mathematical equations correspond to the code variables `delta_w_h`, `delta_b_h`, `delta_w_out`, and `delta_b_out` in `NeuralNetMLP`.

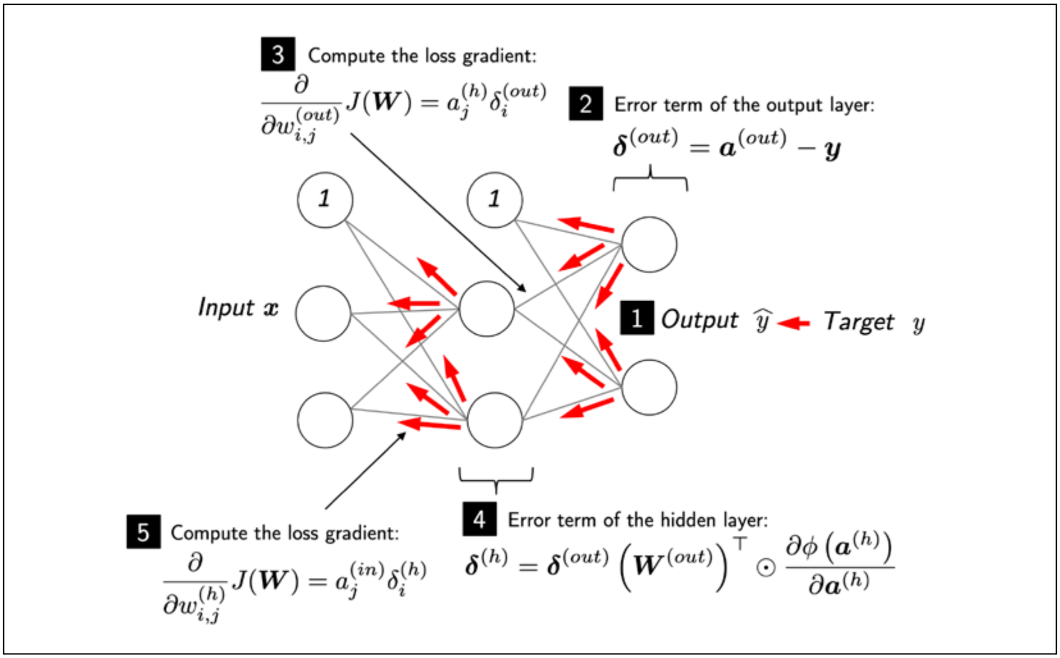
Lastly, after we have computed the gradients, we can update the weights by taking an opposite step toward the gradient for each layer *l*:

$$\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta \Delta^{(l)}$$

This is implemented as follows:

```
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out
```

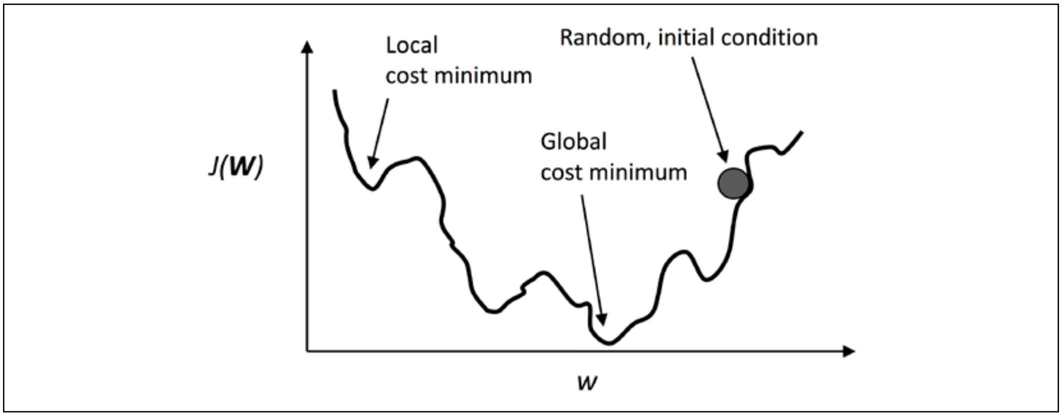
To bring everything together, let's summarize backpropagation in the following figure:


3 Compute the loss gradient:
$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\boldsymbol{W}) = a_j^{(h)} \delta_i^{(out)}$
2 Error term of the output layer:
$\boldsymbol{\delta}^{(out)} = \boldsymbol{a}^{(out)} - \boldsymbol{y}$
1
1
Input $\boldsymbol{x}$
1 Output $\hat{y}$
Target y
5 Compute the loss gradient:
$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\boldsymbol{W}) = a_j^{(in)} \delta_i^{(h)}$
4 Error term of the hidden layer:
$\boldsymbol{\delta}^{(h)} = \boldsymbol{\delta}^{(out)} \left(\boldsymbol{W}^{(out)}\right)^\top \odot \frac{\partial \phi\left(\boldsymbol{a}^{(h)}\right)}{\partial \boldsymbol{a}^{(h)}}$

About the convergence in neural networks

You might be wondering why we did not use regular gradient descent but instead used mini-batch learning to train our NN for the handwritten digit classification. You may recall our discussion on SGD that we used to implement online learning. In online learning, we compute the gradient based on a single training example (*k* = 1) at a time to perform the weight update. Although this is a stochastic approach, it often leads to very accurate solutions with a much faster convergence than regular gradient descent. Mini-batch learning is a special form of SGD where we compute the gradient based on a subset *k* of the *n* training examples with 1 < *k* < *n*. Mini-batch learning has the advantage over online learning that we can make use of our vectorized implementations to improve computational efficiency. However, we can update the weights much faster than in regular gradient descent. Intuitively, you can think of mini-batch learning as predicting the voter turnout of a presidential election from a poll by asking only a representative subset of the population rather than asking the entire population (which would be equal to running the actual election).

Multilayer NNs are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines. In multilayer NNs, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface and the optimization algorithm can easily become trapped in local minima, as shown in the following figure:



Note that this representation is extremely simplified since our NN has many dimensions; it makes it impossible to visualize the actual cost surface for the human eye. Here, we only show the cost surface for a single weight on the *x*-axis. However, the main message is that we do not want our algorithm to get trapped in local minima. By increasing the learning rate, we can more readily escape such local minima. On the other hand, we also increase the chance of overshooting the global optimum if the learning rate is too large. Since we initialize the weights randomly, we start with a solution to the optimization problem that is typically hopelessly wrong.

A few last words about the neural network implementation

You may be wondering why we went through all of this theory just to implement a simple multilayer artificial network that can classify handwritten digits instead of using an open source Python machine learning library. In fact, we will introduce more complex NN models in the next chapters, which we will train using the open source TensorFlow library (`https://www.tensorflow.org`).

Although the from-scratch implementation in this chapter seems a bit tedious at first, it was a good exercise for understanding the basics behind backpropagation and NN training, and a basic understanding of algorithms is crucial for applying machine learning techniques appropriately and successfully.

Now that you have learned how feedforward NNs work, we are ready to explore more sophisticated DNNs by using TensorFlow, which allows us to construct NNs more efficiently, as we will see in *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*.

Over the past two years, since its release in November 2015, TensorFlow has gained a lot of popularity among machine learning researchers, who use it to construct DNNs because of its ability to optimize mathematical expressions for computations on multidimensional arrays utilizing **graphics processing units** (**GPUs**). While TensorFlow can be considered a low-level deep learning library, simplifying APIs such as Keras have been developed that make the construction of common deep learning models even more convenient, which we will see in *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*.

Summary

In this chapter, you have learned the basic concepts behind multilayer artificial NNs, which are currently the hottest topic in machine learning research. In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, we started our journey with simple single-layer NN structures and now we have connected multiple neurons to a powerful NN architecture to solve complex problems such as handwritten digit recognition. We demystified the popular backpropagation algorithm, which is one of the building blocks of many NN models that are used in deep learning. After learning about the backpropagation algorithm in this chapter, we are well equipped for exploring more complex DNN architectures. In the remaining chapters, we will cover TensorFlow, an open source library geared toward deep learning, which allows us to implement and train multilayer NNs more efficiently.

13

Parallelizing Neural Network Training with TensorFlow

In this chapter, we will move on from the mathematical foundations of machine learning and deep learning to focus on TensorFlow. TensorFlow is one of the most popular deep learning libraries currently available, and it lets us implement neural networks (NNs) much more efficiently than any of our previous NumPy implementations. In this chapter, we will start using TensorFlow and see how it brings significant benefits to training performance.

This chapter will begin the next stage of our journey into machine learning and deep learning, and we will explore the following topics:

- How TensorFlow improves training performance
- Working with TensorFlow's `Dataset` API (`tf.data`) to build input pipelines and efficient model training
- Working with TensorFlow to write optimized machine learning code
- Using TensorFlow high-level APIs to build a multilayer NN
- Choosing activation functions for artificial NNs
- Introducing Keras (`tf.keras`), a high-level wrapper around TensorFlow that can be used to implement common deep learning architectures conveniently

TensorFlow and training performance

TensorFlow can speed up our machine learning tasks significantly. To understand how it can do this, let's begin by discussing some of the performance challenges we typically run into when we run expensive calculations on our hardware. Then, we will take a high-level look at what TensorFlow is and what our learning approach will be in this chapter.

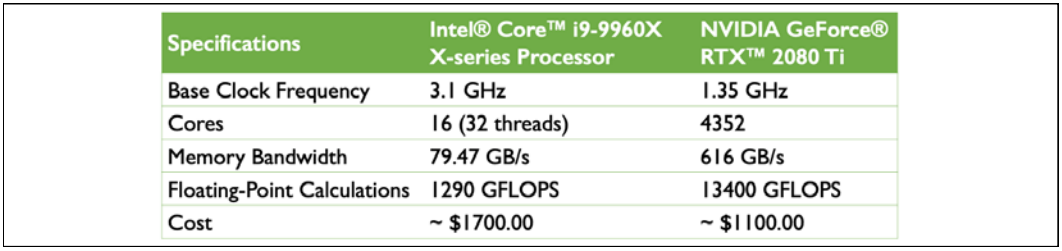
Performance challenges

The performance of computer processors has, of course, been continuously improving in recent years, and that allows us to train more powerful and complex learning systems, which means that we can improve the predictive performance of our machine learning models. Even the cheapest desktop computer hardware that's available right now comes with processing units that have multiple cores.

In the previous chapters, we saw that many functions in scikit-learn allow us to spread those computations over multiple processing units. However, by default, Python is limited to execution on one core due to the **global interpreter lock** (**GIL**). So, although we, indeed, take advantage of Python's multiprocessing library to distribute our computations over multiple cores, we still have to consider that the most advanced desktop hardware rarely comes with more than eight or 16 such cores.

You will recall from *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*, that we implemented a very simple multilayer perceptron (MLP) with only one hidden layer consisting of 100 units. We had to optimize approximately 80,000 weight parameters ([784*100 + 100] + [100 * 10] + 10 = 79,510) to learn a model for a very simple image classification task. The images in MNIST are rather small (28×28), and we can only imagine the explosion in the number of parameters if we wanted to add additional hidden layers or work with images that have higher pixel densities. Such a task would quickly become unfeasible for a single processing unit. The question then becomes, how can we tackle such problems more effectively?

The obvious solution to this problem is to use graphics processing units (GPUs), which are real work horses. You can think of a graphics card as a small computer cluster inside your machine. Another advantage is that modern GPUs are relatively cheap compared to the state-of-the-art central processing units (CPUs), as you can see in the following overview:



Specifications	Intel® Core™ i9-9960X X-series Processor	NVIDIA GeForce® RTX™ 2080 Ti
Base Clock Frequency	3.1 GHz	1.35 GHz
Cores	16 (32 threads)	4352
Memory Bandwidth	79.47 GB/s	616 GB/s
Floating-Point Calculations	1290 GFLOPS	13400 GFLOPS
Cost	~ $1700.00	~ $1100.00

The sources for the information in the table are the following websites (Date: October 2019):

- `https://ark.intel.com/content/www/us/en/ark/products/189123/intel-core-i9-9960x-x-series-processor-22m-cache-up-to-4-50-ghz.html`
- `https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/`

At 65 percent of the price of a modern CPU, we can get a GPU that has 272 times more cores and is capable of around 10 times more floating-point calculations per second. So, what is holding us back from utilizing GPUs for our machine learning tasks? The challenge is that writing code to target GPUs is not as simple as executing Python code in our interpreter. There are special packages, such as CUDA and OpenCL, that allow us to target the GPU. However, writing code in CUDA or OpenCL is probably not the most convenient environment for implementing and running machine learning algorithms. The good news is that this is what TensorFlow was developed for!

What is TensorFlow?

TensorFlow is a scalable and multiplatform programming interface for implementing and running machine learning algorithms, including convenience wrappers for deep learning. TensorFlow was developed by the researchers and engineers from the Google Brain team. While the main development is led by a team of researchers and software engineers at Google, its development also involves many contributions from the open source community. TensorFlow was initially built for internal use at Google, but it was subsequently released in November 2015 under a permissive open source license. Many machine learning researchers and practitioners from academia and industry have adapted TensorFlow to develop deep learning solutions.

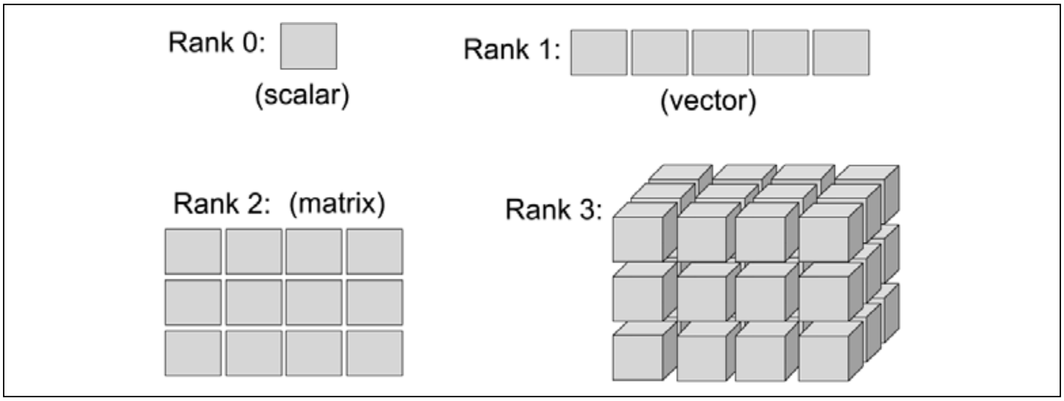
To improve the performance of training machine learning models, TensorFlow allows execution on both CPUs and GPUs. However, its greatest performance capabilities can be discovered when using GPUs. TensorFlow supports CUDA-enabled GPUs officially. Support for OpenCL-enabled devices is still experimental. However, OpenCL will likely be officially supported in the near future. TensorFlow currently supports frontend interfaces for a number of programming languages.

Luckily for us as Python users, TensorFlow's Python API is currently the most complete API, thereby it attracts many machine learning and deep learning practitioners. Furthermore, TensorFlow has an official API in C++. In addition, new tools based on TensorFlow have been released, TensorFlow.js and TensorFlow Lite, that focus on running and deploying machine learning models in a web browser and on mobile and Internet of Things (IoT) devices. The APIs in other languages, such as Java, Haskell, Node.js, and Go, are not stable yet, but the open source community and TensorFlow developers are constantly improving them.

TensorFlow is built around a computation graph composed of a set of nodes. Each node represents an operation that may have zero or more input or output. A tensor is created as a symbolic handle to refer to the input and output of these operations.

Mathematically, tensors can be understood as a generalization of scalars, vectors, matrices, and so on. More concretely, a scalar can be defined as a rank-0 tensor, a vector can be defined as a rank-1 tensor, a matrix can be defined as a rank-2 tensor, and matrices stacked in a third dimension can be defined as rank-3 tensors. But note that in TensorFlow, the values are stored in NumPy arrays, and the tensors provide references to these arrays.

To make the concept of a tensor clearer, consider the following figure, which represents tensors of ranks 0 and 1 in the first row, and tensors of ranks 2 and 3 in the second row:



In the original TensorFlow release, TensorFlow computations relied on constructing a static, directed graph to represent the data flow. As the use of static computation graphs proved to be a major friction point for many users, the TensorFlow library recently received a major overhaul with its 2.0 version, which makes building and training NN models a lot simpler. While TensorFlow 2.0 still supports static computation graphs, it now uses dynamic computation graphs, which allows for more flexibility.

How we will learn TensorFlow

First, we are going to cover TensorFlow's programming model, in particular, creating and manipulating tensors. Then, we will see how to load data and utilize TensorFlow `Dataset` objects, which will allow us to iterate through a dataset efficiently. In addition, we will discuss the existing, ready-to-use datasets in the `tensorflow_datasets` submodule and learn how to use them.

After learning about these basics, the `tf.keras` API will be introduced and we will move forward to building machine learning models, learn how to compile and train the models, and learn how to save the trained models on disk for future evaluation.

First steps with TensorFlow

In this section, we will take our first steps in using the low-level TensorFlow API. After installing TensorFlow, we will cover how to create tensors in TensorFlow and different ways of manipulating them, such as changing their shape, data type, and so on.

Installing TensorFlow

Depending on how your system is set up, you can typically just use Python's `pip` installer and install TensorFlow from PyPI by executing the following from your terminal:

```
pip install tensorflow
```

This will install the latest *stable* version, which is 2.0.0 at the time of writing. In order to ensure that the code presented in this chapter can be executed as expected, it is recommended that you use TensorFlow 2.0.0, which can be installed by specifying the version explicitly:

```
pip install tensorflow==[desired-version]
```

In case you want to use GPUs (recommended), you need a compatible NVIDIA graphics card, along with the CUDA Toolkit and the NVIDIA cuDNN library to be installed. If your machine satisfies these requirements, you can install TensorFlow with GPU support, as follows:

```
pip install tensorflow-gpu
```

For more information about the installation and setup process, please see the official recommendations at `https://www.tensorflow.org/install/gpu`.

Note that TensorFlow is still under active development; therefore, every couple of months, newer versions are released with significant changes. At the time of writing this chapter, the latest TensorFlow version is 2.0. You can verify your TensorFlow version from your terminal, as follows:

```
python -c 'import tensorflow as tf; print(tf.__version__)'
```

> **Troubleshooting your installation of TensorFlow**
>
> If you experience problems with the installation procedure, read more about system- and platform-specific recommendations that are provided at `https://www.tensorflow.org/install/`. Note that all the code in this chapter can be run on your CPU; using a GPU is entirely optional but recommended if you want to fully enjoy the benefits of TensorFlow. For example, while training some NN models on CPU could take a week, the same models could be trained in just a few hours on a modern GPU. If you have a graphics card, refer to the installation page to set it up appropriately. In addition, you may find this TensorFlow-GPU setup guide helpful, which explains how to install the NVIDIA graphics card drivers, CUDA, and cuDNN on Ubuntu (not required but recommended requirements for running TensorFlow on a GPU): `https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf`. Furthermore, as you will see in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*, you can also train your models using a GPU for free via Google Colab.

Creating tensors in TensorFlow

Now, let's consider a few different ways of creating tensors, and then see some of their properties and how to manipulate them. Firstly, we can simply create a tensor from a list or a NumPy array using the `tf.convert_to_tensor` function as follows:

```
    >>> import tensorflow as tf
```

```
>>> import numpy as np
>>> np.set_printoptions(precision=3)

>>> a = np.array([1, 2, 3], dtype=np.int32)
>>> b = [4, 5, 6]

>>> t_a = tf.convert_to_tensor(a)
>>> t_b = tf.convert_to_tensor(b)

>>> print(t_a)
>>> print(t_b)

tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([4 5 6], shape=(3,), dtype=int32)
```

This resulted in tensors `t_a` and `t_b`, with their properties, `shape=(3,)` and `dtype=int32`, adopted from their source. Similar to NumPy arrays, we can further see these properties:

```
>>> t_ones = tf.ones((2, 3))
>>> t_ones.shape
TensorShape([2, 3])
```

To get access to the values that a tensor refers to, we can simply call the `.numpy()` method on a tensor:

```
>>> t_ones.numpy()
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

Finally, creating a tensor of constant values can be done as follows:

```
>>> const_tensor = tf.constant([1.2, 5, np.pi],
...                             dtype=tf.float32)
>>> print(const_tensor)
tf.Tensor([1.2   5.    3.142], shape=(3,), dtype=float32)
```

Manipulating the data type and shape of a tensor

Learning ways to manipulate tensors is necessary to make them compatible for input to a model or an operation. In this section, you will learn how to manipulate tensor data types and shapes via several TensorFlow functions that cast, reshape, transpose, and squeeze.

The `tf.cast()` function can be used to change the data type of a tensor to a desired type:

```
>>> t_a_new = tf.cast(t_a, tf.int64)
>>> print(t_a_new.dtype)
<dtype: 'int64'>
```

As you will see in upcoming chapters, certain operations require that the input tensors have a certain number of dimensions (that is, rank) associated with a certain number of elements (shape). Thus, we might need to change the shape of a tensor, add a new dimension, or squeeze an unnecessary dimension. TensorFlow provides useful functions (or operations) to achieve this, such as `tf.transpose()`, `tf.reshape()`, and `tf.squeeze()`. Let's take a look at some examples:

- **Transposing a tensor**:

  ```
  >>> t = tf.random.uniform(shape=(3, 5))
  >>> t_tr = tf.transpose(t)
  >>> print(t.shape, ' --> ', t_tr.shape)
  (3, 5)  -->  (5, 3)
  ```

- **Reshaping a tensor (for example, from a 1D vector to a 2D array)**:

  ```
  >>> t = tf.zeros((30,))
  >>> t_reshape = tf.reshape(t, shape=(5, 6))
  >>> print(t_reshape.shape)
  (5, 6)
  ```

- **Removing the unnecessary dimensions (dimensions that have size 1, which are not needed)**:

  ```
  >>> t = tf.zeros((1, 2, 1, 4, 1))
  >>> t_sqz = tf.squeeze(t, axis=(2, 4))
  >>> print(t.shape, ' --> ', t_sqz.shape)
  (1, 2, 1, 4, 1)  -->  (1, 2, 4)
  ```

Applying mathematical operations to tensors

Applying mathematical operations, in particular linear algebra operations, is necessary for building most machine learning models. In this subsection, we will cover some widely used linear algebra operations, such as element-wise product, matrix multiplication, and computing the norm of a tensor.

First, let's instantiate two random tensors, one with uniform distribution in the range [–1, 1) and the other with a standard normal distribution:

```
>>> tf.random.set_seed(1)
```

```
>>> t1 = tf.random.uniform(shape=(5, 2),
...                        minval=-1.0, maxval=1.0)

>>> t2 = tf.random.normal(shape=(5, 2),
...                       mean=0.0, stddev=1.0)
```

Notice that `t1` and `t2` have the same shape. Now, to compute the element-wise product of `t1` and `t2`, we can use the following:

```
>>> t3 = tf.multiply(t1, t2).numpy()
>>> print(t3)
[[-0.27  -0.874]
 [-0.017 -0.175]
 [-0.296 -0.139]
 [-0.727  0.135]
 [-0.401  0.004]]
```

To compute the mean, sum, and standard deviation along a certain axis (or axes), we can use `tf.math.reduce_mean()`, `tf.math.reduce_sum()`, and `tf.math.reduce_std()`. For example, the mean of each column in `t1` can be computed as follows:

```
>>> t4 = tf.math.reduce_mean(t1, axis=0)
>>> print(t4)
tf.Tensor([0.09  0.207], shape=(2,), dtype=float32)
```

The matrix-matrix product between `t1` and `t2` (that is, $t_1 \times t_2^{\mathrm{T}}$, where the superscript T is for transpose) can be computed by using the `tf.linalg.matmul()` function as follows:

```
>>> t5 = tf.linalg.matmul(t1, t2, transpose_b=True)
>>> print(t5.numpy())
[[-1.144  1.115 -0.87  -0.321  0.856]
 [ 0.248 -0.191  0.25  -0.064 -0.331]
 [-0.478  0.407 -0.436  0.022  0.527]
 [ 0.525 -0.234  0.741 -0.593 -1.194]
 [-0.099  0.26   0.125 -0.462 -0.396]]
```

On the other hand, computing $t_1 \times t_2^{\mathrm{T}}$ is performed by transposing `t1`, resulting in an array of size 2×2:

```
>>> t6 = tf.linalg.matmul(t1, t2, transpose_a=True)
>>> print(t6.numpy())
[[-1.711  0.302]
 [ 0.371 -1.049]]
```

Finally, the tf.norm() function is useful for computing the L^p norm of a tensor. For example, we can calculate the L^2 norm of t1 as follows:

```
>>> norm_t1 = tf.norm(t1, ord=2, axis=1).numpy()
>>> print(norm_t1)
[1.046 0.293 0.504 0.96  0.383]
```

To verify that this code snippet computes the L^2 norm of t1 correctly, you can compare the results with the following NumPy function: np.sqrt(np.sum(np.square(t1), axis=1)).

Split, stack, and concatenate tensors

In this subsection, we will cover TensorFlow operations for splitting a tensor into multiple tensors, or the reverse: stacking and concatenating multiple tensors into a single one.

Assume that we have a single tensor and we want to split it into two or more tensors. For this, TensorFlow provides a convenient tf.split() function, which divides an input tensor into a list of equally-sized tensors. We can determine the desired number of splits as an integer using the argument num_or_size_splits to split a tensor along a desired dimension specified by the axis argument. In this case, the total size of the input tensor along the specified dimension must be divisible by the desired number of splits. Alternatively, we can provide the desired sizes in a list. Let's have a look at an example of both these options:

- **Providing the number of splits (must be divisible)**:

  ```
  >>> tf.random.set_seed(1)
  >>> t = tf.random.uniform((6,))
  >>> print(t.numpy())
  [0.165 0.901 0.631 0.435 0.292 0.643]

  >>> t_splits = tf.split(t, num_or_size_splits=3)
  >>> [item.numpy() for item in t_splits]
  [array([0.165, 0.901], dtype=float32),
   array([0.631, 0.435], dtype=float32),
   array([0.292, 0.643], dtype=float32)]
  ```

  In this example, a tensor of size 6 was divided into a list of three tensors each with size 2.

- **Providing the sizes of different splits**:

  Alternatively, instead of defining the number of splits, we can also specify the sizes of the output tensors directly. Here, we are splitting a tensor of size 5 into tensors of sizes 3 and 2:

```
>>> tf.random.set_seed(1)
>>> t = tf.random.uniform((5,))
>>> print(t.numpy())
[0.165 0.901 0.631 0.435 0.292]

>>> t_splits = tf.split(t, num_or_size_splits=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.165, 0.901, 0.631], dtype=float32),
 array([0.435, 0.292], dtype=float32)]
```

Sometimes, we are working with multiple tensors and need to concatenate or stack them to create a single tensor. In this case, TensorFlow functions such as `tf.stack()` and `tf.concat()` come in handy. For example, let's create a 1D tensor, `A`, containing 1s with size `3` and a 1D tensor, `B`, containing 0s with size `2` and concatenate them into a 1D tensor, `C`, of size `5`:

```
>>> A = tf.ones((3,))
>>> B = tf.zeros((2,))
>>> C = tf.concat([A, B], axis=0)
>>> print(C.numpy())
[1. 1. 1. 0. 0.]
```

If we create 1D tensors `A` and `B`, both with size `3`, then we can stack them together to form a 2D tensor, `S`:

```
>>> A = tf.ones((3,))
>>> B = tf.zeros((3,))
>>> S = tf.stack([A, B], axis=1)
>>> print(S.numpy())
[[1. 0.]
 [1. 0.]
 [1. 0.]]
```

The TensorFlow API has many operations that you can use for building a model, processing your data, and more. However, covering every function is outside the scope of this book, where we will focus on the most essential ones. For the full list of operations and functions, you can refer to the documentation page of TensorFlow at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf`.

Building input pipelines using tf.data – the TensorFlow Dataset API

When we are training a deep NN model, we usually train the model incrementally using an iterative optimization algorithm such as stochastic gradient descent, as we have seen in previous chapters.

As mentioned at the beginning of this chapter, the Keras API is a wrapper around TensorFlow for building NN models. The Keras API provides a method, `.fit()`, for training the models. In cases where the training dataset is rather small and can be loaded as a tensor into the memory, TensorFlow models (that are built with the Keras API) can directly use this tensor via their `.fit()` method for training. In typical use cases, however, when the dataset is too large to fit into the computer memory, we will need to load the data from the main storage device (for example, the hard drive or solid-state drive) in chunks, that is, batch by batch (note the use of the term "batch" instead of "mini-batch" in this chapter to stay close to the TensorFlow terminology). In addition, we may need to construct a data-processing pipeline to apply certain transformations and preprocessing steps to our data, such as mean centering, scaling, or adding noise to augment the training procedure and to prevent overfitting.

Applying preprocessing functions manually every time can be quite cumbersome. Luckily, TensorFlow provides a special class for constructing efficient and convenient preprocessing pipelines. In this section, we will see an overview of different methods for constructing a TensorFlow `Dataset`, including dataset transformations and common preprocessing steps.

Creating a TensorFlow Dataset from existing tensors

If the data already exists in the form of a tensor object, a Python list, or a NumPy array, we can easily create a dataset using the `tf.data.Dataset.from_tensor_slices()` function. This function returns an object of class `Dataset`, which we can use to iterate through the individual elements in the input dataset. As a simple example, consider the following code, which creates a dataset from a list of values:

```
>>> a = [1.2, 3.4, 7.5, 4.1, 5.0, 1.0]
>>> ds = tf.data.Dataset.from_tensor_slices(a)
>>> print(ds)
<TensorSliceDataset shapes: (), types: tf.float32>
```

We can easily iterate through a dataset entry by entry as follows:

```
>>> for item in ds:
...     print(item)
tf.Tensor(1.2, shape=(), dtype=float32)
tf.Tensor(3.4, shape=(), dtype=float32)
tf.Tensor(7.5, shape=(), dtype=float32)
tf.Tensor(4.1, shape=(), dtype=float32)
tf.Tensor(5.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
```

If we want to create batches from this dataset, with a desired batch size of 3, we can do this as follows:

```
>>> ds_batch = ds.batch(3)
>>> for i, elem in enumerate(ds_batch, 1):
...    print('batch {}:'.format(i), elem.numpy())
batch 1: [1.2 3.4 7.5]
batch 2: [4.1 5.  1. ]
```

This will create two batches from this dataset, where the first three elements go into batch #1, and the remaining elements go into batch #2. The .batch() method has an optional argument, drop_remainder, which is useful for cases when the number of elements in the tensor is not divisible by the desired batch size. The default for drop_remainder is False. We will see more examples illustrating the behavior of this method later in the subsection *Shuffle, batch, and repeat*.

Combining two tensors into a joint dataset

Often, we may have the data in two (or possibly more) tensors. For example, we could have a tensor for features and a tensor for labels. In such cases, we need to build a dataset that combines these tensors together, which will allow us to retrieve the elements of these tensors in tuples.

Assume that we have two tensors, t_x and t_y. Tensor t_x holds our feature values, each of size 3, and t_y stores the class labels. For this example, we first create these two tensors as follows:

```
>>> tf.random.set_seed(1)
>>> t_x = tf.random.uniform([4, 3], dtype=tf.float32)
>>> t_y = tf.range(4)
```

Now, we want to create a joint dataset from these two tensors. Note that there is a required one-to-one correspondence between the elements of these two tensors:

```
>>> ds_x = tf.data.Dataset.from_tensor_slices(t_x)
>>> ds_y = tf.data.Dataset.from_tensor_slices(t_y)
>>>
>>> ds_joint = tf.data.Dataset.zip((ds_x, ds_y))
>>> for example in ds_joint:
...     print('  x:', example[0].numpy(),
...           '  y:', example[1].numpy())

  x: [0.165 0.901 0.631]   y: 0
  x: [0.435 0.292 0.643]   y: 1
  x: [0.976 0.435 0.66 ]   y: 2
  x: [0.605 0.637 0.614]   y: 3
```

Here, we first created two separate datasets, namely ds_x and ds_y. We then used the zip function to form a joint dataset. Alternatively, we can create the joint dataset using tf.data.Dataset.from_tensor_slices() as follows:

```
>>> ds_joint = tf.data.Dataset.from_tensor_slices((t_x, t_y))
>>> for example in ds_joint:
...     print('  x:', example[0].numpy(),
...           '  y:', example[1].numpy())

  x: [0.165 0.901 0.631]   y: 0
  x: [0.435 0.292 0.643]   y: 1
  x: [0.976 0.435 0.66 ]   y: 2
  x: [0.605 0.637 0.614]   y: 3
```

which results in the same output.

Note that a common source of error could be that the element-wise correspondence between the original features (*x*) and labels (*y*) might be lost (for example, if the two datasets are shuffled separately). However, once they are merged into one dataset, it is safe to apply these operations.

Next, we will see how to apply transformations to each individual element of a dataset. For this, we will use the previous ds_joint dataset and apply feature-scaling to scale the values to the range [-1, 1), as currently the values of t_x are in the range [0, 1) based on a random uniform distribution:

```
>>> ds_trans = ds_joint.map(lambda x, y: (x*2-1.0, y))
>>> for example in ds_trans:
...    print('  x:', example[0].numpy(),
...          '  y:', example[1].numpy())

  x: [-0.67   0.803  0.262]   y: 0
  x: [-0.131 -0.416  0.285]   y: 1
  x: [ 0.952 -0.13   0.32 ]   y: 2
  x: [ 0.21   0.273  0.229]   y: 3
```

Applying this sort of transformation can be used for a user-defined function. For example, if we have a dataset created from the list of image filenames on disk, we can define a function to load the images from these filenames and apply that function by calling the .map() method. You will see an example of applying multiple transformations to a dataset later in this chapter.

Shuffle, batch, and repeat

As was mentioned in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification,* to train an NN model using stochastic gradient descent optimization, it is important to feed training data as randomly shuffled batches. You have already seen how to create batches by calling the `.batch()` method of a dataset object. Now, in addition to creating batches, you will see how to shuffle and reiterate over the datasets. We will continue working with the previous `ds_joint` dataset.

First, let's create a shuffled version from the `ds_joint` dataset:

```
>>> tf.random.set_seed(1)
>>> ds = ds_joint.shuffle(buffer_size=len(t_x))
>>> for example in ds:
...     print('  x:', example[0].numpy(),
...           '  y:', example[1].numpy())

  x: [0.976 0.435 0.66 ]   y: 2
  x: [0.435 0.292 0.643]   y: 1
  x: [0.165 0.901 0.631]   y: 0
  x: [0.605 0.637 0.614]   y: 3
```

where the rows are shuffled without losing the one-to-one correspondence between the entries in `x` and `y`. The `.shuffle()` method requires an argument called `buffer_size`, which determines how many elements in the dataset are grouped together before shuffling. The elements in the buffer are randomly retrieved and their place in the buffer is given to the next elements in the original (unshuffled) dataset. Therefore, if we choose a small `buffer_size`, we may not shuffle the dataset perfectly.

If the dataset is small, choosing a relatively small `buffer_size` may negatively affect the predictive performance of the NN as the dataset may not be completely randomized. In practice, however, it usually does not have a noticeable effect when working with relatively large datasets, which is common in deep learning. Alternatively, to ensure complete randomization during each epoch, we can simply choose a buffer size that is equal to the number of the training examples, as in the preceding code (`buffer_size=len(t_x)`).

You will recall that dividing a dataset into batches for model training is done by calling the `.batch()` method. Now, let's create such batches from the `ds_joint` dataset and take a look at what a batch looks like:

```
>>> ds = ds_joint.batch(batch_size=3,
...                     drop_remainder=False)
>>> batch_x, batch_y = next(iter(ds))
>>> print('Batch-x:\n', batch_x.numpy())
```

```
Batch-x:
[[0.165 0.901 0.631]
 [0.435 0.292 0.643]
 [0.976 0.435 0.66 ]]

>>> print('Batch-y: ', batch_y.numpy())
Batch-y: [0 1 2]
```

In addition, when training a model for multiple epochs, we need to shuffle and iterate over the dataset by the desired number of epochs. So, let's repeat the batched dataset twice:

```
>>> ds = ds_joint.batch(3).repeat(count=2)
>>> for i,(batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())

0 (3, 3) [0 1 2]
1 (1, 3) [3]
2 (3, 3) [0 1 2]
3 (1, 3) [3]
```

This results in two copies of each batch. If we change the order of these two operations, that is, first batch and then repeat, the results will be different:

```
>>> ds = ds_joint.repeat(count=2).batch(3)
>>> for i,(batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())
0 (3, 3) [0 1 2]
1 (3, 3) [3 0 1]
2 (2, 3) [2 3]
```

Notice the difference between the batches. When we first batch and then repeat, we get four batches. On the other hand, when repeat is performed first, three batches are created.

Finally, to get a better understanding of how these three operations (batch, shuffle, and repeat) behave, let's experiment with them in different orders. First, we will combine the operations in the following order: (1) shuffle, (2) batch, and (3) repeat:

```
## Order 1: shuffle -> batch -> repeat
>>> tf.random.set_seed(1)
>>> ds = ds_joint.shuffle(4).batch(2).repeat(3)
>>> for i,(batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())
0 (2, 3) [2 1]
1 (2, 3) [0 3]
```

```
2 (2, 3) [0 3]
3 (2, 3) [1 2]
4 (2, 3) [3 0]
5 (2, 3) [1 2]
```

Now, let's try a different order: (2) batch, (1) shuffle, and (3) repeat:

```
## Order 2: batch -> shuffle -> repeat
>>> tf.random.set_seed(1)
>>> ds = ds_joint.batch(2).shuffle(4).repeat(3)
>>> for i,(batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())
0 (2, 3) [0 1]
1 (2, 3) [2 3]
2 (2, 3) [0 1]
3 (2, 3) [2 3]
4 (2, 3) [2 3]
5 (2, 3) [0 1]
```

While the first code example (shuffle, batch, repeat) appears to have shuffled the dataset as expected, we can see that in the second case (batch, shuffle, repeat), the elements within a batch were not shuffled at all. We can observe this lack of shuffling by taking a closer look at the tensor containing the target values, `y`. All batches contain either the pair of values `[y=0, y=1]` or the remaining pair of values `[y=2, y=3]`; we do not observe the other possible permutations: `[y=2, y=0]`, `[y=1, y=3]`, and so forth. Note that in order to ensure these results are not coincidental, you may want to repeat this with a higher number than 3. For example, try it with `.repeat(20)`.

Now, can you predict what will happen if we use the shuffle operation after repeat, for example, (2) batch, (3) repeat, (1) shuffle? Give it a try.

> One common source of error is to call `.batch()` twice in a row on a given dataset. By doing this, retrieving items from the resulting dataset will create a batch of batches of examples. Basically, each time you call `.batch()` on a dataset, it will increase the rank of the retrieved tensors by one.

Creating a dataset from files on your local storage disk

In this section, we will build a dataset from image files stored on disk. There is an image folder associated with the online content of this chapter. After downloading the folder, you should be able to see six images of cats and dogs in JPEG format.

This small dataset will show how building a dataset from stored files generally works. To accomplish this, we are going to use two additional modules in TensorFlow: `tf.io` to read the image file contents, and `tf.image` to decode the raw contents and image resizing.

tf.io and tf.image modules

The `tf.io` and `tf.image` modules provide a lot of additional and useful functions, which are beyond the scope of the book. You are encouraged to browse through the official documentation to learn more about these functions:

`https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/io` for `tf.io`

`https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/image` for `tf.image`

Before we start, let's take a look at the content of these files. We will use the `pathlib` library to generate a list of image files:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
...                     imgdir_path.glob('*.jpg')])
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', 'cat_dog_
images/cat-02.jpg', 'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-
01.jpg', 'cat_dog_images/dog-02.jpg']
```

Next, we will visualize these image examples using Matplotlib:

```
>>> import matplotlib.pyplot as plt

>>> fig = plt.figure(figsize=(10, 5))
>>> for i, file in enumerate(file_list):
...     img_raw = tf.io.read_file(file)
...     img = tf.image.decode_image(img_raw)
...     print('Image shape: ', img.shape)
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(img)
...     ax.set_title(os.path.basename(file), size=15)
>>> plt.tight_layout()
>>> plt.show()
Image shape:  (900, 1200, 3)
Image shape:  (900, 1200, 3)
Image shape:  (900, 1200, 3)
Image shape:  (900, 742, 3)
Image shape:  (800, 1200, 3)
Image shape:  (800, 1200, 3)
```

The following figure shows the example images:

cat_dog_images/cat-01.jpg
cat_dog_images/cat-02.jpg
cat_dog_images/cat-03.jpg
cat_dog_images/dog-01.jpg
cat_dog_images/dog-02.jpg
cat_dog_images/dog-03.jpg

Just from this visualization and the printed image shapes, we can already see that the images have different aspect ratios. If you print the aspect ratios (or data array shapes) of these images, you will see that some images are 900 pixels high and 1200 pixels wide (900×1200), some are 800×1200, and one is 900×742. Later, we will preprocess these images to a consistent size. Another point to consider is that the labels for these images are provided within their filenames. So, we extract these labels from the list of filenames, assigning label `1` to dogs and label `0` to cats:

```
>>> labels = [1 if 'dog' in os.path.basename(file) else 0
...           for file in file_list]
>>> print(labels)
[1, 0, 0, 0, 1, 1]
```

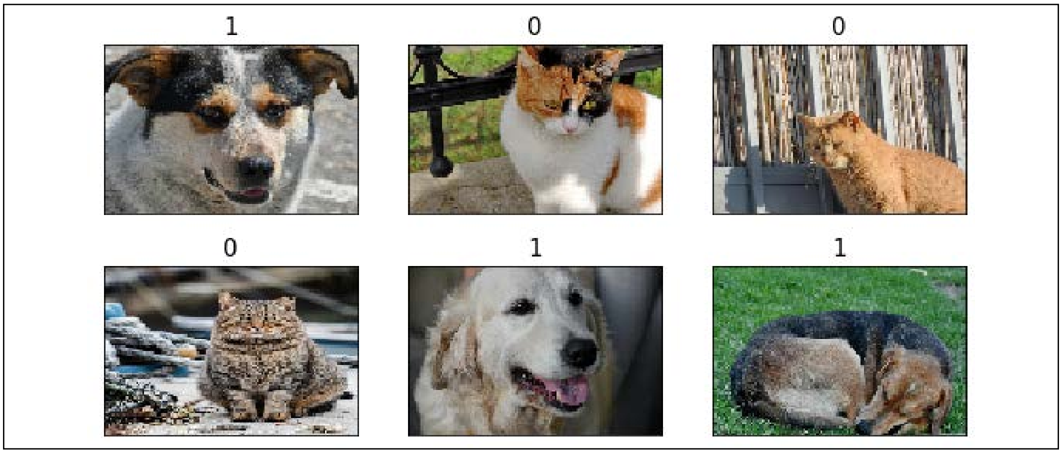
Now, we have two lists: a list of filenames (or paths of each image) and a list of their labels. In the previous section, you already learned two ways of creating a joint dataset from two tensors. Here, we will use the second approach as follows:

```
>>> ds_files_labels = tf.data.Dataset.from_tensor_slices(
...                                   (file_list, labels))
>>> for item in ds_files_labels:
...     print(item[0].numpy(), item[1].numpy())
b'cat_dog_images/dog-03.jpg' 1
b'cat_dog_images/cat-01.jpg' 0
b'cat_dog_images/cat-02.jpg' 0
b'cat_dog_images/cat-03.jpg' 0
b'cat_dog_images/dog-01.jpg' 1
b'cat_dog_images/dog-02.jpg' 1
```

We have called this dataset `ds_files_labels`, since it has filenames and labels. Next, we need to apply transformations to this dataset: load the image content from its file path, decode the raw content, and resize it to a desired size, for example, 80 × 120. Previously, we saw how to apply a lambda function using the `.map()` method. However, since we need to apply multiple preprocessing steps this time, we are going to write a helper function instead and use it when calling the `.map()` method:

```
>>> def load_and_preprocess(path, label):
...     image = tf.io.read_file(path)
...     image = tf.image.decode_jpeg(image, channels=3)
...     image = tf.image.resize(image, [img_height, img_width])
...     image /= 255.0
...     return image, label

>>> img_width, img_height = 120, 80
>>> ds_images_labels = ds_files_labels.map(load_and_preprocess)
>>>
>>> fig = plt.figure(figsize=(10, 6))
>>> for i,example in enumerate(ds_images_labels):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0])
...     ax.set_title('{}'.format(example[1].numpy()),
...                  size=15)
>>> plt.tight_layout()
>>> plt.show()
```

This results in the following visualization of the retrieved example images, along with their labels:



The load_and_preprocess() function wraps all four steps into a single function, including the loading of the raw content, decoding it, and resizing the images. The function then returns a dataset that we can iterate over and apply other operations that we learned about in the previous sections.

Fetching available datasets from the tensorflow_datasets library

The tensorflow_datasets library provides a nice collection of freely available datasets for training or evaluating deep learning models. The datasets are nicely formatted and come with informative descriptions, including the format of features and labels and their type and dimensionality, as well as the citation of the original paper that introduced the dataset in BibTeX format. Another advantage is that these datasets are all prepared and ready to use as tf.data.Dataset objects, so all the functions we covered in the previous sections can be used directly. So, let's see how to use these datasets in action.

First, we need to install the tensorflow_datasets library via pip from the command line:

```
pip install tensorflow-datasets
```

Now, let's import this module and take a look at the list of available datasets:

```
>>> import tensorflow_datasets as tfds
>>> print(len(tfds.list_builders()))
101
>>> print(tfds.list_builders()[:5])
['abstract_reasoning', 'aflw2k3d', 'amazon_us_reviews', 'bair_robot_
pushing_small', 'bigearthnet']
```

The preceding code indicates that there are currently 101 datasets available (101 datasets at the time of writing this chapter, but this number will likely increase)—we printed the first five datasets to the command line. There are two ways of fetching a dataset, which we will cover in the following paragraphs by fetching two different datasets: CelebA (celeb_a) and the MNIST digit dataset.

The first approach consists of three steps:

1. Calling the dataset builder function
2. Executing the download_and_prepare() method
3. Calling the as_dataset() method

Let's work with the first step for the CelebA dataset and print the associated description that is provided within the library:

```
>>> celeba_bldr = tfds.builder('celeb_a')

>>> print(celeba_bldr.info.features)
FeaturesDict({'image': Image(shape=(218, 178, 3), dtype=tf.uint8),
'landmarks': FeaturesDict({'lefteye_x': Tensor(shape=(), dtype=tf.
int64), 'lefteye_y': Tensor(shape=(), dtype=tf.int64), 'righteye_x':
Tensor(shape=(), dtype=tf.int64), 'righteye_y': ...

>>> print(celeba_bldr.info.features['image'])
Image(shape=(218, 178, 3), dtype=tf.uint8)

>>> print(celeba_bldr.info.features['attributes'].keys())
dict_keys(['5_o_Clock_Shadow', 'Arched_Eyebrows', ...

>>> print(celeba_bldr.info.citation)
@inproceedings{conf/iccv/LiuLWT15,
  added-at = {2018-10-09T00:00:00.000+0200},
  author = {Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang,
Xiaoou},
  biburl = {https://www.bibsonomy.org/bibtex/250e4959be61db325d2f02c1d
8cd7bfbb/dblp},
  booktitle = {ICCV},
  crossref = {conf/iccv/2015},
  ee = {http://doi.ieeecomputersociety.org/10.1109/ICCV.2015.425},
  interhash = {3f735aaa11957e73914bbe2ca9d5e702},
  intrahash = {50e4959be61db325d2f02c1d8cd7bfbb},
  isbn = {978-1-4673-8391-2},
  keywords = {dblp},
  pages = {3730-3738},
  publisher = {IEEE Computer Society},
  timestamp = {2018-10-11T11:43:28.000+0200},
  title = {Deep Learning Face Attributes in the Wild.},
  url = {http://dblp.uni-trier.de/db/conf/iccv/iccv2015.
html#LiuLWT15},
  year = 2015
}
```

This provides some useful information to understand the structure of this dataset. The features are stored as a dictionary with three keys: `'image'`, `'landmarks'`, and `'attributes'`.

The 'image' entry refers to the face image of a celebrity; 'landmarks' refers to the dictionary of extracted facial points, such as the position of the eyes, nose, and so on; and 'attributes' is a dictionary of 40 facial attributes for the person in the image, like facial expression, makeup, hair properties, and so on.

Next, we will call the download_and_prepare() method. This will download the data and store it on disk in a designated folder for all TensorFlow Datasets. If you have already done this once, it will simply check whether the data is already downloaded so that it does not re-download it if it already exists in the designated location:

```
>>> celeba_bldr.download_and_prepare()
```

Next, we will instantiate the datasets as follows:

```
>>> datasets = celeba_bldr.as_dataset(shuffle_files=False)
>>> datasets.keys()
dict_keys(['test', 'train', 'validation'])
```

This dataset is already split into train, test, and validation datasets. In order to see what the image examples look like, we can execute the following code:

```
>>> ds_train = datasets['train']
>>> assert isinstance(ds_train, tf.data.Dataset)

>>> example = next(iter(ds_train))
>>> print(type(example))
<class 'dict'>
>>> print(example.keys())
dict_keys(['image', 'landmarks', 'attributes'])
```

Note that the elements of this dataset come in a dictionary. If we want to pass this dataset to a supervised deep learning model during training, we have to reformat it as a tuple of (features, label). For the label, we will use the 'Male' category from the attributes. We will do this by applying a transformation via map():

```
>>> ds_train = ds_train.map(lambda item:
...                 (item['image'],
...                 tf.cast(item['attributes']['Male'], tf.int32)))
```

Finally, let's batch the dataset and take a batch of 18 examples from it to visualize them with their labels:

```
>>> ds_train = ds_train.batch(18)
>>> images, labels = next(iter(ds_train))
>>> print(images.shape, labels)
(18, 218, 178, 3) tf.Tensor([0 0 0 1 1 1 0 1 1 0 1 1 0 1 0 1 1 1],
shape=(18,), dtype=int32)

>>> fig = plt.figure(figsize=(12, 8))
>>> for i,(image,label) in enumerate(zip(images, labels)):
...     ax = fig.add_subplot(3, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image)
...     ax.set_title('{}'.format(label), size=15)
>>> plt.show()
```

The examples and their labels that are retrieved from `ds_train` are shown in the following figure:

This was all we needed to do to fetch and use the CelebA image dataset.

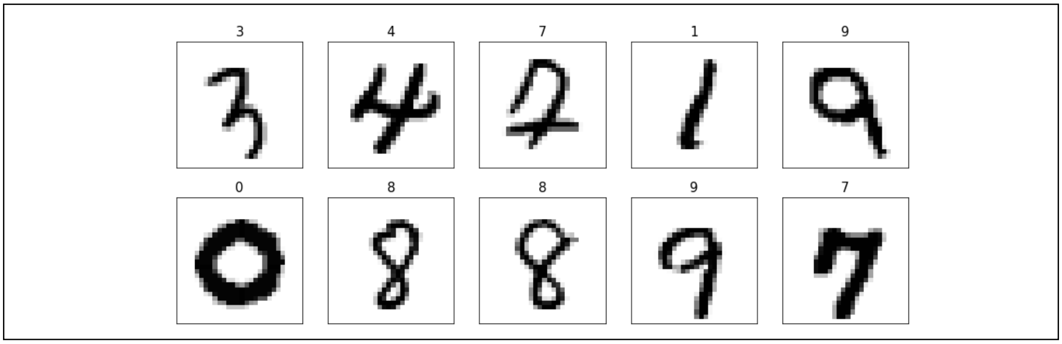
Next, we will proceed with the second approach for fetching a dataset from `tensorflow_datasets`. There is a wrapper function called `load()` that combines the three steps for fetching a dataset in one. Let's see how it can be used to fetch the MNIST digit dataset:

```
>>> mnist, mnist_info = tfds.load('mnist', with_info=True,
...                               shuffle_files=False)
>>> print(mnist_info)
tfds.core.DatasetInfo(
    name='mnist',
    version=1.0.0,
    description='The MNIST database of handwritten digits.',
    urls=['https://storage.googleapis.com/cvdf-datasets/mnist/'],
    features=FeaturesDict({
        'image': Image(shape=(28, 28, 1), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10)
    },
    total_num_examples=70000,
    splits={
        'test': <tfds.core.SplitInfo num_examples=10000>,
        'train': <tfds.core.SplitInfo num_examples=60000>
    },
    supervised_keys=('image', 'label'),
    citation="""
        @article{lecun2010mnist,
          title={MNIST handwritten digit database},
          author={LeCun, Yann and Cortes, Corinna and Burges, CJ},
          journal={ATT Labs [Online]. Availablist},
          volume={2},
          year={2010}
        }

    """,
    redistribution_info=,
)

>>> print(mnist.keys())
dict_keys(['test', 'train'])
```

As we can see, the MNIST dataset is split into two partitions. Now, we can take the train partition, apply a transformation to convert the elements from a dictionary to a tuple, and visualize 10 examples:

```
>>> ds_train = mnist['train']
>>> ds_train = ds_train.map(lambda item:
...                          (item['image'], item['label']))
>>> ds_train = ds_train.batch(10)
>>> batch = next(iter(ds_train))
>>> print(batch[0].shape, batch[1])
(10, 28, 28, 1) tf.Tensor([8 4 7 7 0 9 0 3 3 3], shape=(10,),
dtype=int64)

>>> fig = plt.figure(figsize=(15, 6))
>>> for i,(image,label) in enumerate(zip(batch[0], batch[1])):
...     ax = fig.add_subplot(2, 5, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image[:, :, 0], cmap='gray_r')
...     ax.set_title('{}'.format(label), size=15)
>>> plt.show()
```

The retrieved example handwritten digits from this dataset are shown as follows:



This concludes our coverage of building and manipulating datasets and fetching datasets from the `tensorflow_datasets` library. Next, we will see how to build NN models in TensorFlow.

TensorFlow style guide

Note that the official TensorFlow style guide (`https://www.tensorflow.org/community/style_guide`) recommends using two-character spacing for code indents. However, this book uses four characters for indents as it is more consistent with the official Python style guide and also helps with displaying the code syntax highlighting in many text editors correctly, as well as the accompanying Jupyter code notebooks at `https://github.com/rasbt/python-machine-learning-book-3rd-edition`.

Building an NN model in TensorFlow

So far in this chapter, you have learned about the basic utility components of TensorFlow for manipulating tensors and organizing data into formats that we can iterate over during training. In this section, we will finally implement our first predictive model in TensorFlow. As TensorFlow is a bit more flexible but also more complex than machine learning libraries such as scikit-learn, we will start with a simple linear regression model.

The TensorFlow Keras API (tf.keras)

Keras is a high-level NN API and was originally developed to run on top of other libraries such as TensorFlow and Theano. Keras provides a user-friendly and modular programming interface that allows easy prototyping and the building of complex models in just a few lines of code. Keras can be installed independently from PyPI and then configured to use TensorFlow as its backend engine. Keras is tightly integrated into TensorFlow and its modules are accessible through `tf.keras`. In TensorFlow 2.0, `tf.keras` has become the primary and recommended approach for implementing models. This has the advantage that it supports TensorFlow-specific functionalities, such as dataset pipelines using `tf.data`, which you learned about in the previous section. In this book, we will use the `tf.keras` module to build NN models.

As you will see in the following subsections, the Keras API (`tf.keras`) makes building an NN model extremely easy. The most commonly used approach for building an NN in TensorFlow is through `tf.keras.Sequential()`, which allows stacking layers to form a network. A stack of layers can be given in a Python list to a model defined as `tf.keras.Sequential()`. Alternatively, the layers can be added one by one using the `.add()` method.

Furthermore, `tf.keras` allows us to define a model by subclassing `tf.keras.Model`. This gives us more control over the forward pass by defining the `call()` method for our model class to specify the forward pass explicitly. We will see examples of both of these approaches for building an NN model using the `tf.keras` API.

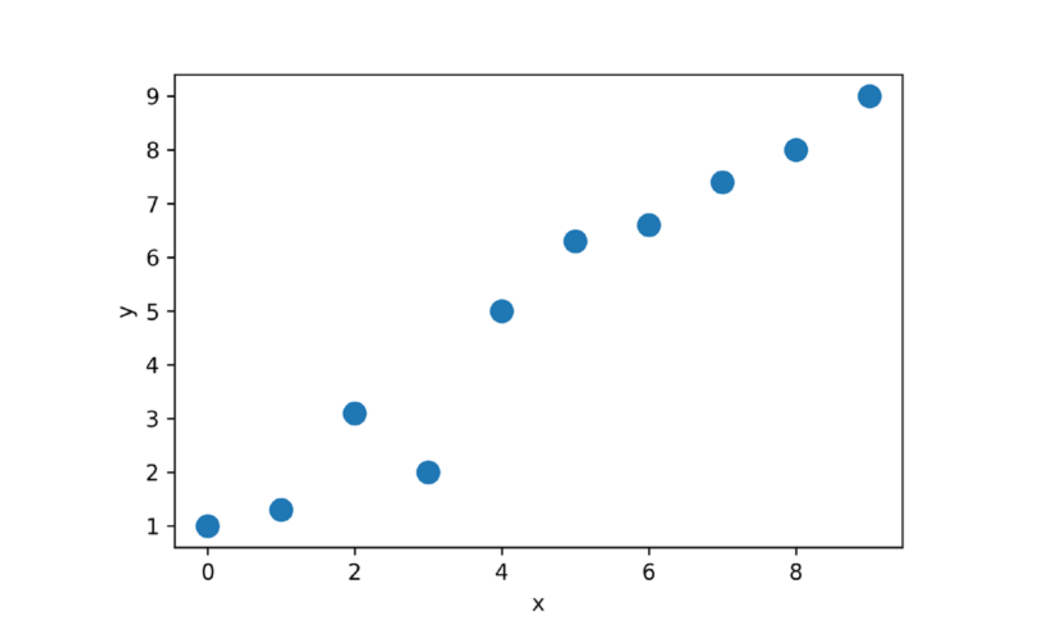
Finally, as you will see in the following subsections, models built using the `tf.keras` API can be compiled and trained via the `.compile()` and `.fit()` methods.

Building a linear regression model

In this subsection, we will build a simple model to solve a linear regression problem. First, let's create a toy dataset in NumPy and visualize it:

```
>>> X_train = np.arange(10).reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3,
...                     6.6, 7.4, 8.0, 9.0])

>>> plt.plot(X_train, y_train, 'o', markersize=10)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

As a result, the training examples will be shown in a scatterplot as follows:



Next, we will standardize the features (mean centering and dividing by the standard deviation) and create a TensorFlow `Dataset`:

```
>>> X_train_norm = (X_train - np.mean(X_train))/np.std(X_train)
>>> ds_train_orig = tf.data.Dataset.from_tensor_slices(
...                   (tf.cast(X_train_norm, tf.float32),
...                    tf.cast(y_train, tf.float32)))
```

Now, we can define our model for linear regression as $z = wx + b$. Here, we are going to use the Keras API. `tf.keras` provides predefined layers for building complex NN models, but to start, you will learn how to define a model from scratch. Later in this chapter, you will see how to use those predefined layers.

For this regression problem, we will define a new class derived from the `tf.keras.Model` class. Subclassing `tf.keras.Model` allows us to use the Keras tools for exploring a model, training, and evaluation. In the constructor of our class, we will define the parameters of our model, `w` and `b`, which correspond to the weight and the bias parameters, respectively. Finally, we will define the `call()` method to determine how this model uses the input data to generate its output:

```
>>> class MyModel(tf.keras.Model):
...     def __init__(self):
...         super(MyModel, self).__init__()
...         self.w = tf.Variable(0.0, name='weight')
...         self.b = tf.Variable(0.0, name='bias')
...
...     def call(self, x):
...         return self.w * x + self.b
```

Next, we will instantiate a new model from the `MyModel()` class that we can train based on the training data. The TensorFlow Keras API provides a method named `.summary()` for models that are instantiated from `tf.keras.Model`, which allows us to get a summary of the model components layer by layer and the number of parameters in each layer. Since we have sub-classed our model from `tf.keras.Model`, the `.summary()` method is also available to us. But, in order to be able to call `model.summary()`, we first need to specify the dimensionality of the input (the number of features) to this model. We can do this by calling `model.build()` with the expected shape of the input data:

```
>>> model = MyModel()
>>> model.build(input_shape=(None, 1))
>>> model.summary()
Model: "my_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

Note that we used `None` as a placeholder for the first dimension of the expected input tensor via `model.build()`, which allows us to use an arbitrary batch size. However, the number of features is fixed (here `1`) as it directly corresponds to the number of weight parameters of the model. Building model layers and parameters after instantiation by calling the `.build()` method is called *late variable creation*. For this simple model, we already created the model parameters in the constructor; therefore, specifying the `input_shape` via `build()` has no further effect on our parameters, but still it is needed if we want to call `model.summary()`.

After defining the model, we can define the cost function that we want to minimize to find the optimal model weights. Here, we will choose the **mean squared error** (**MSE**) as our cost function. Furthermore, to learn the weight parameters of the model, we will use stochastic gradient descent. In this subsection, we will implement this training via the stochastic gradient descent procedure by ourselves, but in the next subsection, we will use the Keras methods `compile()` and `fit()` to do the same thing.

To implement the stochastic gradient descent algorithm, we need to compute the gradients. Rather than manually computing the gradients, we will use the TensorFlow API `tf.GradientTape`. We will cover `tf.GradientTape` and its different behaviors in *Chapter 14, Going Deeper – The Mechanics of TensorFlow*. The code is as follows:

```
>>> def loss_fn(y_true, y_pred):
...     return tf.reduce_mean(tf.square(y_true - y_pred))

>>> def train(model, inputs, outputs, learning_rate):
...     with tf.GradientTape() as tape:
...         current_loss = loss_fn(model(inputs), outputs)
...     dW, db = tape.gradient(current_loss, [model.w, model.b])
...     model.w.assign_sub(learning_rate * dW)
...     model.b.assign_sub(learning_rate * db)
```

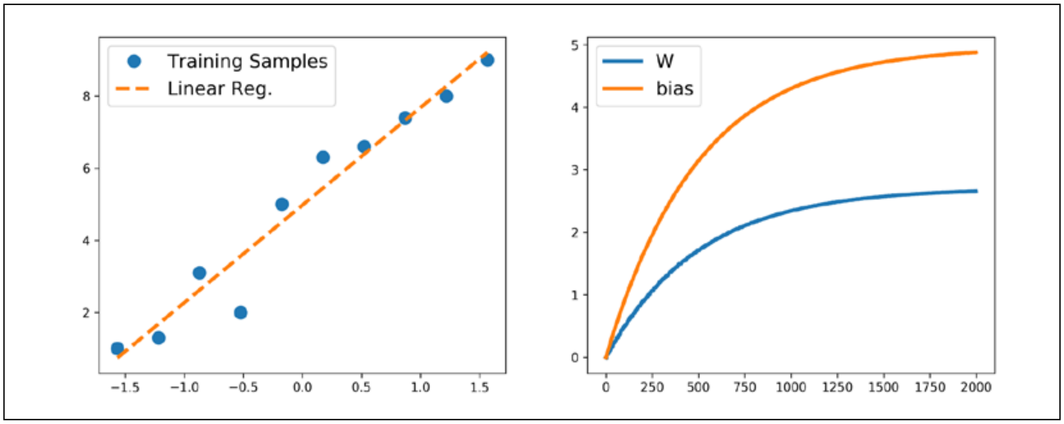
Now, we can set the hyperparameters and train the model for 200 epochs. We will create a batched version of the dataset and repeat the dataset with `count=None`, which will result in an infinitely repeated dataset:

```
>>> tf.random.set_seed(1)
>>> num_epochs = 200
>>> log_steps = 100
>>> learning_rate = 0.001
>>> batch_size = 1
>>> steps_per_epoch = int(np.ceil(len(y_train) / batch_size))

>>> ds_train = ds_train_orig.shuffle(buffer_size=len(y_train))
>>> ds_train = ds_train.repeat(count=None)
>>> ds_train = ds_train.batch(1)
>>> Ws, bs = [], []

>>> for i, batch in enumerate(ds_train):
...     if i >= steps_per_epoch * num_epochs:
...         # break the infinite loop
...         break
...     Ws.append(model.w.numpy())
...     bs.append(model.b.numpy())
...
...     bx, by = batch
...     loss_val = loss_fn(model(bx), by)
...
...     train(model, bx, by, learning_rate=learning_rate)
...     if i%log_steps==0:
...         print('Epoch {:4d} Step {:2d} Loss {:6.4f}'.format(
...                int(i/steps_per_epoch), i, loss_val))

Epoch    0 Step  0 Loss 43.5600
Epoch   10 Step 100 Loss 0.7530
Epoch   20 Step 200 Loss 20.1759
Epoch   30 Step 300 Loss 23.3976
Epoch   40 Step 400 Loss 6.3481
Epoch   50 Step 500 Loss 4.6356
Epoch   60 Step 600 Loss 0.2411
Epoch   70 Step 700 Loss 0.2036
Epoch   80 Step 800 Loss 3.8177
Epoch   90 Step 900 Loss 0.9416
Epoch  100 Step 1000 Loss 0.7035
Epoch  110 Step 1100 Loss 0.0348
Epoch  120 Step 1200 Loss 0.5404
Epoch  130 Step 1300 Loss 0.1170
Epoch  140 Step 1400 Loss 0.1195
Epoch  150 Step 1500 Loss 0.0944
Epoch  160 Step 1600 Loss 0.4670
Epoch  170 Step 1700 Loss 2.0695
Epoch  180 Step 1800 Loss 0.0020
Epoch  190 Step 1900 Loss 0.3612
```

Let's look at the trained model and plot it. For test data, we will create a NumPy array of values evenly spaced between 0 to 9. Since we trained our model with standardized features, we will also apply the same standardization to the test data:

```
>>> print('Final Parameters: ', model.w.numpy(), model.b.numpy())
Final Parameters:  2.6576622 4.8798566

>>> X_test = np.linspace(0, 9, num=100).reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> y_pred = model(tf.cast(X_test_norm, dtype=tf.float32))

>>> fig = plt.figure(figsize=(13, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>>> plt.legend(['Training examples', 'Linear Reg.'], fontsize=15)
>>> ax.set_xlabel('x', size=15)
>>> ax.set_ylabel('y', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(Ws, lw=3)
>>> plt.plot(bs, lw=3)
>>> plt.legend(['Weight w', 'Bias unit b'], fontsize=15)
>>> ax.set_xlabel('Iteration', size=15)
>>> ax.set_ylabel('Value', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

The following figure shows the scatterplot of the training examples and the trained linear regression model, as well as the convergence history of the weight, *w*, and the bias unit, *b*:



Model training via the .compile() and .fit() methods

In the previous example, we saw how to train a model by writing a custom function, `train()`, and applied stochastic gradient descent optimization. However, writing the `train()` function can be a repeatable task across different projects. The TensorFlow Keras API provides a convenient `.fit()` method that can be called on an instantiated model. To show how this works, let's create a new model and compile it by selecting the optimizer, loss function, and evaluation metrics:

```
>>> tf.random.set_seed(1)
>>> model = MyModel()
>>> model.compile(optimizer='sgd',
...               loss=loss_fn,
...               metrics=['mae', 'mse'])
```

Now, we can simply call the `fit()` method to train the model. We can pass a batched dataset (like `ds_train`, which was created in the previous example). However, this time you will see that we can pass the NumPy arrays for `x` and `y` directly, without needing to create a dataset:

```
>>> model.fit(X_train_norm, y_train,
...           epochs=num_epochs, batch_size=batch_size,
...           verbose=1)

Train on 10 samples
Epoch 1/200
10/10 [==============================] - 0s 4ms/sample - loss: 27.8578
- mae: 4.5810 - mse: 27.8578
Epoch 2/200
10/10 [==============================] - 0s 738us/sample - loss:
18.6640 - mae: 3.7395 - mse: 18.6640
...
Epoch 200/200
10/10 [==============================] - 0s 1ms/sample - loss: 0.4139
- mae: 0.4942 - mse: 0.4139
```

After the model is trained, visualize the results and make sure that they are similar to the results of the previous method.

Building a multilayer perceptron for classifying flowers in the Iris dataset

In the previous example, you saw how to build a model from scratch. We trained this model using stochastic gradient descent optimization. While we started our journey based on the simplest possible example, you can see that defining the model from scratch, even for such a simple case, is neither appealing nor a good practice. TensorFlow instead provides already defined layers through `tf.keras.layers` that can be readily used as the building blocks of an NN model. In this section, you will learn how to use these layers to solve a classification task using the Iris flower dataset and build a two-layer perceptron using the Keras API. First, let's get the data from `tensorflow_datasets`:

```
>>> iris, iris_info = tfds.load('iris', with_info=True)
>>> print(iris_info)
```

This prints some information about this dataset (not printed here to save space). However, you will notice in the shown information that this dataset comes with only one partition, so we have to split the dataset into training and testing partitions (and validation for proper machine learning practice) on our own. Let's assume that we want to use two-thirds of the dataset for training and keep the remaining examples for testing. The `tensorflow_datasets` library provides a convenient tool that allows us to determine slices and splits via the `DatasetBuilder` object prior to loading a dataset. You can find out more about splits at `https://www.tensorflow.org/datasets/splits`.

An alternative approach is to load the entire dataset first and then use `.take()` and `.skip()` to split the dataset to two partitions. If the dataset is not shuffled at first, we can also shuffle the dataset. However, we need to be very careful with this because it can lead to mixing the train/test examples, which is not acceptable in machine learning. To avoid this, we have to set an argument, `reshuffle_each_iteration=False`, in the `.shuffle()` method. The code for splitting the dataset into train/test is as follows:

```
>>> tf.random.set_seed(1)
>>> ds_orig = iris['train']
>>> ds_orig = ds_orig.shuffle(150, reshuffle_each_iteration=False)

>>> ds_train_orig = ds_orig.take(100)
>>> ds_test = ds_orig.skip(100)
```

Next, as you have already seen in the previous sections, we need to apply a transformation via the `.map()` method to convert the dictionary to a tuple:

```
>>> ds_train_orig = ds_train_orig.map(
...     lambda x: (x['features'], x['label']))

>>> ds_test = ds_test.map(
...     lambda x: (x['features'], x['label']))
```

Now, we are ready to use the Keras API to build a model efficiently. In particular, using the `tf.keras.Sequential` class, we can stack a few Keras layers and build an NN. You can see the list of all the Keras layers that are already available at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers`. For this problem, we are going to use the `Dense` layer (`tf.keras.layers.Dense`), which is also known as a fully connected (FC) layer or linear layer, and can be best represented by $f(w \times x + b)$, where x is the input features, w and b are the weight matrix and the bias vector, and f is the activation function.

If you think of the layers in an NN, each layer receives its inputs from the preceding layer; therefore, its dimensionality (rank and shape) is fixed. Typically, we need to concern ourselves with the dimensionality of output only when we design an NN architecture. (Note: the first layer is the exception, but TensorFlow/Keras allows us to decide the input dimensionality of the first layer after defining the model through *late variable creation*.) Here, we want to define a model with two hidden layers. The first one receives an input of four features and projects them to 16 neurons. The second layer receives the output of the previous layer (which has size `16`) and projects them to three output neurons, since we have three class labels. This can be done using the `Sequential` class and the `Dense` layer in Keras as follows:

```
>>> iris_model = tf.keras.Sequential([
...         tf.keras.layers.Dense(16, activation='sigmoid',
...                               name='fc1', input_shape=(4,)),
...         tf.keras.layers.Dense(3, name='fc2',
...                               activation='softmax')])

>>> iris_model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
fc1 (Dense)                  (None, 16)                80
_________________________________________________________________
fc2 (Dense)                  (None, 3)                 51
=================================================================
Total params: 131
Trainable params: 131
Non-trainable params: 0
_________________________________________________________________
```

Notice that we determined the input shape for the first layer via `input_shape=(4,)`, and therefore, we did not have to call `.build()` anymore in order to use `iris_model.summary()`.

The printed model summary indicates that the first layer (`fc1`) has 80 parameters, and the second layer has 51 parameters. You can verify that by $(n_{in} + 1) \times n_{out}$, where n_{in} is the number of input units, and n_{out} is the number of output units. Recall that for a fully (densely) connected layer, the learnable parameters are the weight matrix of size $n_{in} \times n_{out}$ and the bias vector of size n_{out}. Furthermore, notice that we used the sigmoid activation function for the first layer and softmax activation for the last (output) layer. Softmax activation in the last layer is used to support multi-class classification, since here we have three class labels (which is why we have three neurons at the output layer). We will discuss the different activation functions and their applications later in this chapter.

Next, we will compile this model to specify the loss function, the optimizer, and the metrics for evaluation:

```
>>> iris_model.compile(optimizer='adam',
...                    loss='sparse_categorical_crossentropy',
...                    metrics=['accuracy'])
```

Now, we can train the model. We will specify the number of epochs to be `100` and the batch size to be `2`. In the following code, we will build an infinitely repeating dataset, which will be passed to the `fit()` method for training the model. In this case, in order for the `fit()` method to be able to keep track of the epochs, it needs to know the number of steps for each epoch.

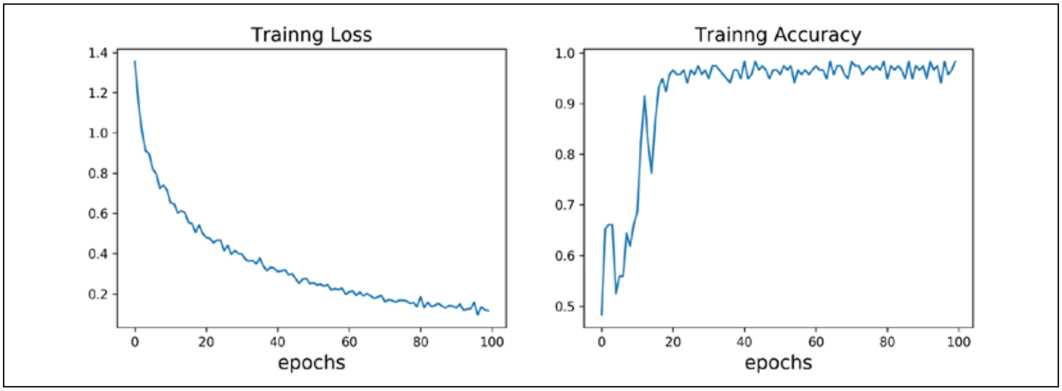
Given the size of our training data (here, `100`) and the batch size (`batch_size`), we can determine the number of steps in each epoch, `steps_per_epoch`:

```
>>> num_epochs = 100
>>> training_size = 100
>>> batch_size = 2
>>> steps_per_epoch = np.ceil(training_size / batch_size)

>>> ds_train = ds_train_orig.shuffle(buffer_size=training_size)
>>> ds_train = ds_train.repeat()
>>> ds_train = ds_train.batch(batch_size=batch_size)
>>> ds_train = ds_train.prefetch(buffer_size=1000)

>>> history = iris_model.fit(ds_train, epochs=num_epochs,
...                          steps_per_epoch=steps_per_epoch,
...                          verbose=0)
```

The returned variable history keeps the training loss and the training accuracy (since they were specified as metrics to `iris_model.compile()`) after each epoch. We can use this to visualize the learning curves as follows:

```
>>> hist = history.history

>>> fig = plt.figure(figsize=(12, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(hist['loss'], lw=3)
>>> ax.set_title('Training loss', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(hist['accuracy'], lw=3)
>>> ax.set_title('Training accuracy', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

The learning curves (training loss and training accuracy) are as follows:



Evaluating the trained model on the test dataset

Since we specified `'accuracy'` as our evaluation metric in `iris_model.compile()`, we can now directly evaluate the model on the test dataset:

```
>>> results = iris_model.evaluate(ds_test.batch(50), verbose=0)
>>> print('Test loss: {:.4f}   Test Acc.: {:.4f}'.format(*results))
Test loss: 0.0692   Test Acc.: 0.9800
```

Notice that we have to batch the test dataset as well, to ensure that the input to the model has the correct dimension (rank). As we discussed earlier, calling `.batch()` will increase the rank of the retrieved tensors by 1. The input data for `.evaluate()` must have one designated dimension for the batch, although here (for evaluation), the size of the batch does not matter. Therefore, if we pass `ds_batch.batch(50)` to the `.evaluate()` method, the entire test dataset will be processed in one batch of size 50, but if we pass `ds_batch.batch(1)`, 50 batches of size 1 will be processed.

Saving and reloading the trained model

Trained models can be saved on disk for future use. This can be done as follows:

```
>>> iris_model.save('iris-classifier.h5',
...             overwrite=True,
...             include_optimizer=True,
...             save_format='h5')
```

The first option is the filename. Calling `iris_model.save()` will save both the model architecture and all the learned parameters. However, if you want to save only the architecture, you can use the `iris_model.to_json()` method, which saves the model configuration in JSON format. Or if you want to save only the model weights, you can do that by calling `iris_model.save_weights()`. The `save_format` can be specified to be either `'h5'` for the HDF5 format or `'tf'` for TensorFlow format.

Now, let's reload the saved model. Since we have saved both the model architecture and the weights, we can easily rebuild and reload the parameters in just one line:

```
>>> iris_model_new = tf.keras.models.load_model('iris-classifier.h5')
```

Try to verify the model architecture by calling `iris_model_new.summary()`.

Finally, let's evaluate this new model that is reloaded on the test dataset to verify that the results are the same as before:

```
>>> results = iris_model_new.evaluate(ds_test.batch(33), verbose=0)
>>> print('Test loss: {:.4f}   Test Acc.: {:.4f}'.format(*results))
Test loss: 0.0692   Test Acc.: 0.9800
```

Choosing activation functions for multilayer neural networks

For simplicity, we have only discussed the sigmoid activation function in the context of multilayer feedforward NNs so far; we used it in the hidden layer as well as the output layer in the MLP implementation in *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*.

Note that in this book, the sigmoidal logistic function, $\sigma(z) = \frac{1}{1 + e^{-z}}$, is referred to as the *sigmoid* function for brevity, which is common in machine learning literature. In the following subsections, you will learn more about alternative nonlinear functions that are useful for implementing multilayer NNs.

Technically, we can use any function as an activation function in multilayer NNs as long as it is differentiable. We can even use linear activation functions, such as in Adaline (*Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*). However, in practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to introduce nonlinearity in a typical artificial NN to be able to tackle complex problems. The sum of linear functions yields a linear function after all.

The logistic (sigmoid) activation function that we used in *Chapter 12* probably mimics the concept of a neuron in a brain most closely—we can think of it as the probability of whether a neuron fires. However, the logistic (sigmoid) activation function can be problematic if we have highly negative input, since the output of the sigmoid function will be close to zero in this case. If the sigmoid function returns output that is close to zero, the NN will learn very slowly, and it will be more likely to get trapped in the local minima during training. This is why people often prefer a hyperbolic tangent as an activation function in hidden layers.

Before we discuss what a hyperbolic tangent looks like, let's briefly recapitulate some of the basics of the logistic function and look at a generalization that makes it more useful for multilabel classification problems.

Logistic function recap

As was mentioned in the introduction to this section, the logistic function is, in fact, a special case of a sigmoid function. You will recall from the section on logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, that we can use a logistic function to model the probability that sample *x* belongs to the positive class (class `1`) in a binary classification task.

The given net input, *z*, is shown in the following equation:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x$$

The logistic (sigmoid) function will compute the following:

$$\phi_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

Note that w_0 is the bias unit (*y*-axis intercept, which means $x_0 = 1$). To provide a more concrete example, let's take a model for a two-dimensional data point, *x*, and a model with the following weight coefficients assigned to the *w* vector:

```
>>> import numpy as np

>>> X = np.array([1, 1.4, 2.5]) ## first value must be 1
>>> w = np.array([0.4, 0.3, 0.5])

>>> def net_input(X, w):
...     return np.dot(X, w)
```

```
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))

>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)

>>> print('P(y=1|x) = %.3f' % logistic_activation(X, w))
P(y=1|x) = 0.888
```

If we calculate the net input (*z*) and use it to activate a logistic neuron with those particular feature values and weight coefficients, we get a value of `0.888`, which we can interpret as an 88.8 percent probability that this particular sample, *x*, belongs to the positive class.

In *Chapter 12*, we used the one-hot encoding technique to represent multiclass ground truth labels and designed the output layer consisting of multiple logistic activation units. However, as will be demonstrated by the following code example, an output layer consisting of multiple logistic activation units does not produce meaningful, interpretable probability values:

```
>>> # W : array with shape = (n_output_units, n_hidden_units+1)
>>> #     note that the first column are the bias units
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...               [0.2, 0.4, 1.0, 0.2],
...               [0.6, 1.5, 1.2, 0.7]])

>>> # A : data array with shape = (n_hidden_units + 1, n_samples)
>>> #     note that the first column of this array must be 1
>>> A = np.array([[1, 0.1, 0.4, 0.6]])

>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Net Input: \n', Z)
Net Input:
[ 1.78  0.76  1.65]
>>> print('Output Units:\n', y_probas)
Output Units:
[ 0.85569687  0.68135373  0.83889105]
```

As you can see in the output, the resulting values cannot be interpreted as probabilities for a three-class problem. The reason for this is that they do not sum up to 1. However, this is, in fact, not a big concern if we use our model to predict only the class labels and not the class membership probabilities. One way to predict the class label from the output units obtained earlier is to use the maximum value:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Predicted class label: %d' % y_class)
Predicted class label: 0
```

In certain contexts, it can be useful to compute meaningful class probabilities for multiclass predictions. In the next section, we will take a look at a generalization of the logistic function, the `softmax` function, which can help us with this task.

Estimating class probabilities in multiclass classification via the softmax function

In the previous section, you saw how we can obtain a class label using the `argmax` function. Previously, in the section *Building a multilayer perceptron for classifying flowers in the Iris dataset*, we determined `activation='softmax'` in the last layer of the MLP model. The `softmax` function is a soft form of the `argmax` function; instead of giving a single class index, it provides the probability of each class. Therefore, it allows us to compute meaningful class probabilities in multiclass settings (multinomial logistic regression).

In `softmax`, the probability of a particular sample with net input *z* belonging to the *i*th class can be computed with a normalization term in the denominator, that is, the sum of the exponentially weighted linear functions:

$$p(z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$$

To see `softmax` in action, let's code it up in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
[ 0.44668973  0.16107406  0.39223621]

>>> np.sum(y_probas)
1.0
```

As you can see, the predicted class probabilities now sum up to 1, as we would expect. It is also notable that the predicted class label is the same as when we applied the `argmax` function to the logistic output.

It may help to think of the result of the `softmax` function as a *normalized* output that is useful for obtaining meaningful class-membership predictions in multiclass settings. Therefore, when we build a multiclass classification model in TensorFlow, we can use the `tf.keras.activations.softmax()` function to estimate the probabilities of each class membership for an input batch of examples. To see how we can use the `softmax` activation function in TensorFlow, in the following code, we will convert `Z` to a tensor, with an additional dimension reserved for the batch size:

```
>>> import tensorflow as tf

>>> Z_tensor = tf.expand_dims(Z, axis=0)
>>> tf.keras.activations.softmax(Z_tensor)
<tf.Tensor: id=21, shape=(1, 3), dtype=float64,
numpy=array([[0.44668973, 0.16107406, 0.39223621]])>
```

Broadening the output spectrum using a hyperbolic tangent

Another sigmoidal function that is often used in the hidden layers of artificial NNs is the **hyperbolic tangent** (commonly known as **tanh**), which can be interpreted as a rescaled version of the logistic function:

$$\begin{aligned}\phi_{logistic}(z) &= \frac{1}{1+e^{-z}} \\ \phi_{tanh}(z) &= 2\times\phi_{logistic}(2z)-1 = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}\end{aligned}$$

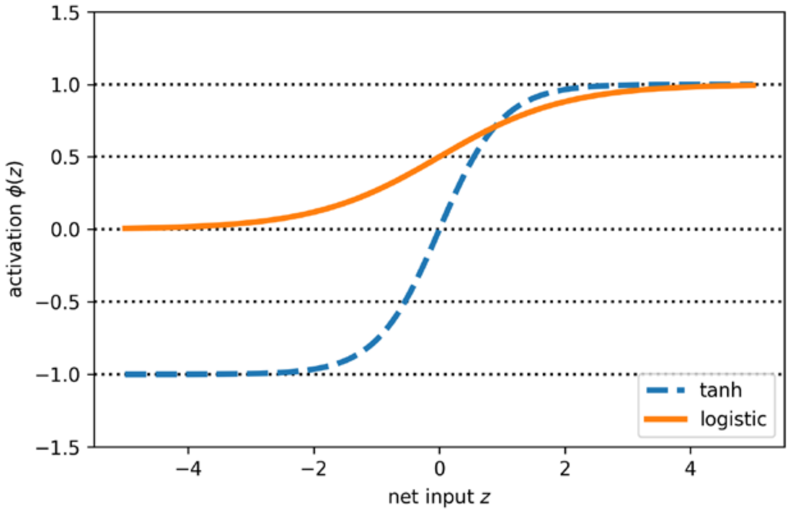
The advantage of the hyperbolic tangent over the logistic function is that it has a broader output spectrum ranging in the open interval (–1, 1), which can improve the convergence of the back-propagation algorithm (*Neural Networks for Pattern Recognition*, *C. M. Bishop*, *Oxford University Press*, pages: 500-501, *1995*).

In contrast, the logistic function returns an output signal ranging in the open interval (0, 1). For a simple comparison of the logistic function and the hyperbolic tangent, let's plot the two sigmoidal functions:

```
>>> import matplotlib.pyplot as plt

>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
```

```
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...          linewidth=3, linestyle='--',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=3,
...          label='logistic')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see, the shapes of the two sigmoidal curves look very similar; however, the `tanh` function has double the output space of the `logistic` function:



Note that we previously implemented the `logistic` and `tanh` functions verbosely for the purpose of illustration. In practice, we can use NumPy's `tanh` function.

Alternatively, when building an NN model, we can use `tf.keras.activations.tanh()` in TensorFlow to achieve the same results:

```
>>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])

>>> tf.keras.activations.tanh(z)
<tf.Tensor: id=14, shape=(2000,), dtype=float64, numpy=
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])>
```

In addition, the logistic function is available in SciPy's `special` module:

```
>>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669,
0.99324034,
       0.99327383])
```

Similarly, we can use the `tf.keras.activations.sigmoid()` function in TensorFlow to do the same computation, as follows:

```
>>> tf.keras.activations.sigmoid(z)
<tf.Tensor: id=16, shape=(2000,), dtype=float64, numpy=
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669,
0.99324034,
       0.99327383])>
```

Rectified linear unit activation

Rectified linear unit (**ReLU**) is another activation function that is often used in deep NNs. Before we delve into ReLU, we should step back and understand the vanishing gradient problem of tanh and logistic activations.

To understand this problem, let's assume that we initially have the net input $z_1 = 20$, which changes to $z_2 = 25$. Computing the tanh activation, we get $\phi(z_1) = 1.0$ and $\phi(z_2) = 1.0$, which shows no change in the output (due to the asymptotic behavior of the tanh function and numerical errors).

This means that the derivative of activations with respect to the net input diminishes as z becomes large. As a result, learning weights during the training phase becomes very slow because the gradient terms may be very close to zero. ReLU activation addresses this issue. Mathematically, ReLU is defined as follows:

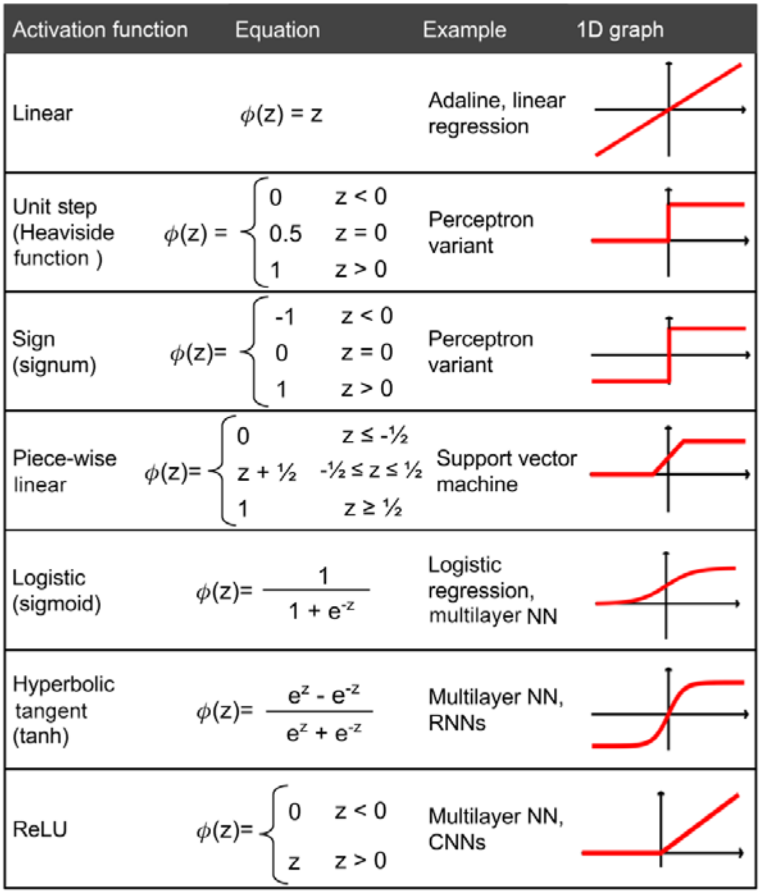
$$\phi(z) = max(0, z)$$

ReLU is still a nonlinear function that is good for learning complex functions with NNs. Besides this, the derivative of ReLU, with respect to its input, is always 1 for positive input values. Therefore, it solves the problem of vanishing gradients, making it suitable for deep NNs. In TensorFlow, we can apply the ReLU activation as follows:

```
>>> tf.keras.activations.tanh(z)
<tf.Tensor: id=23, shape=(2000,), dtype=float64, numpy=array([0.    ,
0.    , 0.    , ..., 4.985, 4.99 , 4.995])>
```

We will use the ReLU activation function in the next chapter as an activation function for multilayer convolutional NNs.

Now that we know more about the different activation functions that are commonly used in artificial NNs, let's conclude this section with an overview of the different activation functions that we encountered in this book:



Activation function	Equation	Example	1D graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit step (Heaviside function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z)= \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise linear	$\phi(z)= \begin{cases} 0 & z \le -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \le z \le \frac{1}{2} \\ 1 & z \ge \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z)= \dfrac{1}{1 + e^{-z}}$	Logistic regression, multilayer NN	
Hyperbolic tangent (tanh)	$\phi(z)= \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z)= \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

You can find the list of all activation functions available in the Keras API at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/activations`.

Summary

In this chapter, you learned how to use TensorFlow, an open source library for numerical computations, with a special focus on deep learning. While TensorFlow is more inconvenient to use than NumPy, due to its additional complexity to support GPUs, it allows us to define and train large, multilayer NNs very efficiently.

Also, you learned about using the TensorFlow Keras API to build complex machine learning and NN models and run them efficiently. We explored model building in TensorFlow by defining a model from scratch via subclassing the `tf.keras.Model` class. Implementing models can be tedious when we have to program at the level of matrix-vector multiplications and define every detail of each operation. However, the advantage is that this allows us, as developers, to combine such basic operations and build more complex models. We then explored `tf.keras.layers`, which makes building NN models a lot easier than implementing them from scratch.

Finally, you learned about different activation functions and understood their behaviors and applications. Specifically, in this chapter, we covered tanh, softmax, and ReLU.

In the next chapter, we'll continue our journey and dive deeper into TensorFlow, where we'll find ourselves working with TensorFlow function decoration and TensorFlow Estimators. Along the way, you'll learn many new concepts, such as variables and feature columns.

14

Going Deeper – The Mechanics of TensorFlow

In *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, we covered how to define and manipulate tensors and worked with the `tf.data` API to build input pipelines. We further built and trained a multilayer perceptron to classify the Iris dataset using the TensorFlow Keras API (`tf.keras`).

Now that we have some hands-on experience with TensorFlow neural network (NN) training and machine learning, it's time to take a deeper dive into the TensorFlow library and explore its rich set of features, which will allow us to implement more advanced deep learning models in upcoming chapters.

In this chapter, we will use different aspects of TensorFlow's API to implement NNs. In particular, we will again use the Keras API, which provides multiple layers of abstraction to make the implementation of standard architectures very convenient. TensorFlow also allows us to implement custom NN layers, which is very useful in research-oriented projects that require more customization. Later in this chapter, we will implement such a custom layer.

To illustrate the different ways of model building using the Keras API, we will also consider the classic **exclusive or** (**XOR**) problem. Firstly, we will build multilayer perceptrons using the `Sequential` class. Then, we will consider other methods, such as subclassing `tf.keras.Model` for defining custom layers. Finally, we will cover `tf.estimator`, a high-level TensorFlow API that encapsulates the machine learning steps from raw input to prediction.

The topics that we will cover are as follows:

- Understanding and working with TensorFlow graphs and migration to TensorFlow v2
- Function decoration for graph compilation

- Working with TensorFlow variables
- Solving the classic XOR problem and understanding model capacity
- Building complex NN models using Keras' `Model` class and the Keras functional API
- Computing gradients using automatic differentiation and `tf.GradientTape`
- Working with TensorFlow Estimators

The key features of TensorFlow

TensorFlow provides us with a scalable, multiplatform programming interface for implementing and running machine learning algorithms. The TensorFlow API has been relatively stable and mature since its 1.0 release in 2017, but it just experienced a major redesign with its recent 2.0 release in 2019, which we are using in this book.

Since its initial release in 2015, TensorFlow has become the most widely adopted deep learning library. However, one of its main friction points was that it was built around static computation graphs. Static computation graphs have certain advantages, such as better graph optimizations behind the scenes and support for a wider range of hardware devices; however, static computation graphs require separate graph declaration and graph evaluation steps, which make it cumbersome for users to develop and work with NNs interactively.

Taking all the user feedback to heart, the TensorFlow team decided to make dynamic computation graphs the default in TensorFlow 2.0, which makes the development and training of NNs much more convenient. In the next section, we will cover some of the important changes from TensorFlow v1.x to v2. Dynamic computation graphs allow for interleaving the graph declaration and graph evaluation steps such that TensorFlow 2.0 feels much more natural for Python and NumPy users compared to previous versions of TensorFlow. However, note that TensorFlow 2.0 still allows users to use the "old" TensorFlow v1.x API via the `tf.compat` submodule. This helps users to transition their code bases more smoothly to the new TensorFlow v2 API.

A key feature of TensorFlow, which was also noted in *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, is its ability to work with single or multiple graphical processing units (GPUs). This allows users to train deep learning models very efficiently on large datasets and large-scale systems.

While TensorFlow is an open source library and can be freely used by everyone, its development is funded and supported by Google. This involves a large team of software engineers who expand and improve the library continuously. Since TensorFlow is an open source library, it also has strong support from other developers outside of Google, who avidly contribute and provide user feedback.

This has made the TensorFlow library more useful to both academic researchers and developers. A further consequence of these factors is that TensorFlow has extensive documentation and tutorials to help new users.

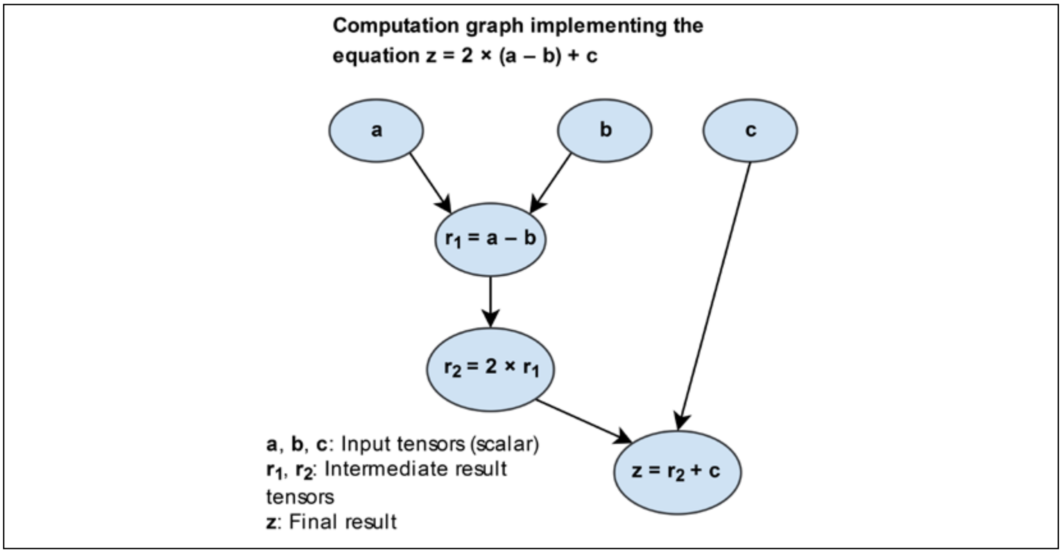
Last, but not least, TensorFlow supports mobile deployment, which also makes it a very suitable tool for production.

TensorFlow's computation graphs: migrating to TensorFlow v2

TensorFlow performs its computations based on a directed acyclic graph (DAG). In TensorFlow v1.x, such graphs could be explicitly defined in the low-level API, although this was not trivial for large and complex models. In this section, we will see how these graphs can be defined for a simple arithmetic computation. Then, we will see how to migrate a graph to TensorFlow v2, the **eager execution** and dynamic graph paradigm, as well as the function decoration for faster computations.

Understanding computation graphs

TensorFlow relies on building a computation graph at its core, and it uses this computation graph to derive relationships between tensors from the input all the way to the output. Let's say that we have rank 0 (scalar) tensors *a*, *b*, and *c* and we want to evaluate $z = 2 \times (a - b) + c$. This evaluation can be represented as a computation graph, as shown in the following figure:


Computation graph implementing the equation z = 2 × (a – b) + c
a
b
c
r1 = a – b
r2 = 2 × r1
z = r2 + c
a, b, c: Input tensors (scalar)
r1, r2: Intermediate result tensors
z: Final result

As you can see, the computation graph is simply a network of nodes. Each node resembles an operation, which applies a function to its input tensor or tensors and returns zero or more tensors as the output. TensorFlow builds this computation graph and uses it to compute the gradients accordingly. In the next subsections, we will see some examples of creating a graph for this computation using TensorFlow v1.x and v2 styles.

Creating a graph in TensorFlow v1.x

In the earlier version of the TensorFlow (v1.x) low-level API, this graph had to be explicitly declared. The individual steps for building, compiling, and evaluating such a computation graph in TensorFlow v1.x are as follows:

1. Instantiate a new, empty computation graph
2. Add nodes (tensors and operations) to the computation graph
3. Evaluate (execute) the graph:
    a. Start a new session
    b. Initialize the variables in the graph
    c. Run the computation graph in this session

Before we take a look at the dynamic approach in TensorFlow v2, let's look at a simple example that illustrates how to create a graph in TensorFlow v1.x for evaluating $z = 2 \times (a - b) + c$, as shown in the previous figure. The variables a, b, and c are scalars (single numbers), and we define these as TensorFlow constants. A graph can then be created by calling `tf.Graph()`. Variables, as well as computations, represent the nodes of the graph, which we will define as follows:

```
## TF v1.x style
>>> g = tf.Graph()

>>> with g.as_default():
...     a = tf.constant(1, name='a')
...     b = tf.constant(2, name='b')
...     c = tf.constant(3, name='c')
...     z = 2*(a-b) + c
```

In this code, we first defined graph `g` via `g=tf.Graph()`. Then, we added nodes to the graph, `g`, using `with g.as_default()`. However, note that if we do not explicitly create a graph, there is always a default graph to which variables and computations will be added automatically.

In TensorFlow v1.x, a session is an environment in which the operations and tensors of a graph can be executed. The `Session` class was removed from TensorFlow v2; However, for the time being, it is still available via the `tf.compat` submodule to allow compatibility with TensorFlow v1.x. A session object can be created by calling `tf.compat.v1.Session()`, which can receive an existing graph (here, `g`) as an argument, as in `Session(graph=g)`.

After launching a graph in a TensorFlow session, we can execute its nodes, that is, evaluate its tensors or execute its operators. Evaluating each individual tensor involves calling its `eval()` method inside the current session. When evaluating a specific tensor in the graph, TensorFlow has to execute all the preceding nodes in the graph until it reaches the given node of interest. In case there are one or more placeholder variables, we also need to provide values for those through the session's `run` method, as we will see later in the chapter.

After defining the static graph in the previous code snippet, we can execute the graph in a TensorFlow session and evaluate the tensor, `z`, as follows:

```
## TF v1.x style
>>> with tf.compat.v1.Session(graph=g) as sess:
...     print(Result: z =', sess.run(z))
Result: z = 1
```

Migrating a graph to TensorFlow v2

Next, let's look at how this code can be migrated to TensorFlow v2. TensorFlow v2 uses dynamic (as opposed to static) graphs by default (this is also called eager execution in TensorFlow), which allows us to evaluate an operation on the fly. Therefore, we do not have to explicitly create a graph and a session, which makes the development workflow much more convenient:

```
## TF v2 style
>>> a = tf.constant(1, name='a')
>>> b = tf.constant(2, name='b')
>>> c = tf.constant(3, name='c')
>>> z = 2*(a - b) + c
>>> tf.print('Result: z= ', z)
Result: z = 1
```

Loading input data into a model: TensorFlow v1.x style

Another important improvement from TensorFlow v1.x to v2 is regarding how data can be loaded into our models. In TensorFlow v2, we can directly feed data in the form of Python variables or NumPy arrays. However, when using the TensorFlow v1.x low-level API, we had to create placeholder variables for providing input data to a model. For the preceding simple computation graph example, $z = 2 \times (a - b) + c$, let's assume that a, b, and c are the input tensors of rank 0. We can then define three placeholders, which we will then use to "feed" data to the model via a so-called feed_dict dictionary, as follows:

```
## TF-v1.x style
>>> g = tf.Graph()
>>> with g.as_default():
...     a = tf.compat.v1.placeholder(shape=None,
...                                  dtype=tf.int32, name='tf_a')
...     b = tf.compat.v1.placeholder(shape=None,
...                                  dtype=tf.int32, name='tf_b')
...     c = tf.compat.v1.placeholder(shape=None,
...                                  dtype=tf.int32, name='tf_c')
...     z = 2*(a-b) + c

>>> with tf.compat.v1.Session(graph=g) as sess:
...     feed_dict={a:1, b:2, c:3}
...     print('Result: z =', sess.run(z, feed_dict=feed_dict))
Result: z = 1
```

Loading input data into a model: TensorFlow v2 style

In TensorFlow v2, all this can simply be done by *defining a regular Python function* with a, b, and c as its input arguments, for example:

```
## TF-v2 style
>>> def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Now, to carry out the computation, we can simply call this function with Tensor objects as function arguments. Note that TensorFlow functions such as add, subtract, and multiply also allow us to provide inputs of higher ranks in the form of a TensorFlow Tensor object, a NumPy array, or possibly other Python objects, such as lists and tuples. In the following code example, we provide scalar inputs (rank 0), as well as rank 1 and rank 2 inputs, as lists:

```
>>> tf.print('Scalar Inputs:', compute_z(1, 2, 3))
Scalar Inputs: 1

>>> tf.print('Rank 1 Inputs:', compute_z([1], [2], [3]))
Rank 1 Inputs: [1]

>>> tf.print('Rank 2 Inputs:', compute_z([[1]], [[2]], [[3]]))
Rank 2 Inputs: [[1]]
```

In this section, you saw how migrating to TensorFlow v2 makes the programming style simple and efficient by avoiding explicit graph and session creation steps. Now that we have seen how TensorFlow v1.x compares to TensorFlow v2, we will focus only on TensorFlow v2 for the remainder of this book. Next, we will take a deeper look into decorating Python functions into a graph that allows for faster computation.

Improving computational performance with function decorators

As you saw in the previous section, we can easily write a normal Python function and utilize TensorFlow operations. However, computations via the eager execution (dynamic graph) mode are not as efficient as the static graph execution in TensorFlow v1.x. Thus, TensorFlow v2 provides a tool called AutoGraph that can automatically transform Python code into TensorFlow's graph code for faster execution. In addition, TensorFlow provides a simple mechanism for compiling a normal Python function to a static TensorFlow graph in order to make the computations more efficient.

To see how this works in practice, let's work with our previous compute_z function and annotate it for graph compilation using the @tf.function decorator:

```
>>> @tf.function
... def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Note that we can use and call this function the same way as before, but now TensorFlow will construct a static graph based on the input arguments. Python supports dynamic typing and polymorphism, so we can define a function such as `def f(a, b): return a+b` and then call it using integer, float, list, or string inputs (recall that `a+b` is a valid operation for lists and strings). While TensorFlow graphs require static types and shapes, `tf.function` supports such a dynamic typing capability. For example, let's call this function with the following inputs:

```
>>> tf.print('Scalar Inputs:', compute_z(1, 2, 3))
>>> tf.print('Rank 1 Inputs:', compute_z([1], [2], [3]))
>>> tf.print('Rank 2 Inputs:', compute_z([[1]], [[2]], [[3]]))
```

This will produce the same outputs as before. Here, TensorFlow uses a tracing mechanism to construct a graph based on the input arguments. For this tracing mechanism, TensorFlow generates a tuple of keys based on the input signatures given for calling the function. The generated keys are as follows:

- For `tf.Tensor` arguments, the key is based on their shapes and dtypes.
- For Python types, such as lists, their `id()` is used to generate cache keys.
- For Python primitive values, the cache keys are based on the input values.

Upon calling such a decorated function, TensorFlow will check whether a graph with the corresponding key has already been generated. If such a graph does not exist, TensorFlow will generate a new graph and store the new key. On the other hand, if we want to limit the way a function can be called, we can specify its input signature via a tuple of `tf.TensorSpec` objects when defining the function. For example, let's redefine the previous function, `compute_z`, and specify that only rank 1 tensors of type `tf.int32` are allowed:

```
>>> @tf.function(input_signature=(tf.TensorSpec(shape=[None],
...                                             dtype=tf.int32),
...                               tf.TensorSpec(shape=[None],
...                                             dtype=tf.int32),
...                               tf.TensorSpec(shape=[None],
...                                             dtype=tf.int32),))
... def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Now, we can call this function using rank 1 tensors (or lists that can be converted to rank 1 tensors):

```
>>> tf.print('Rank 1 Inputs:', compute_z([1], [2], [3]))
>>> tf.print('Rank 1 Inputs:', compute_z([1, 2], [2, 4], [3, 6]))
```

However, calling this function using tensors with ranks other than 1 will result in an error since the rank will not match the specified input signature, as follows:

```
>>> tf.print('Rank 0 Inputs:', compute_z(1, 2, 3)
### will result in error

>>> tf.print('Rank 2 Inputs:', compute_z([[1], [2]],
...                                      [[2], [4]],
...                                      [[3], [6]]))
### will result in error
```

In this section, we learned how to annotate a normal Python function so that TensorFlow will compile it into a graph for faster execution. Next, we will look at TensorFlow variables: how to create them and how to use them.

TensorFlow Variable objects for storing and updating model parameters

We covered `Tensor` objects in *Chapter 13, Parallelizing Neural Network Training with TensorFlow*. In the context of TensorFlow, a `Variable` is a special `Tensor` object that allows us to store and update the parameters of our models during training. A `Variable` can be created by just calling the `tf.Variable` class on user-specified initial values. In the following code, we will generate `Variable` objects of type `float32`, `int32`, `bool`, and `string`:

```
>>> a = tf.Variable(initial_value=3.14, name='var_a')
>>> print(a)
<tf.Variable 'var_a:0' shape=() dtype=float32, numpy=3.14>

>>> b = tf.Variable(initial_value=[1, 2, 3], name='var_b')
>>> print(b)
<tf.Variable 'var_b:0' shape=(3,) dtype=int32, numpy=array([1, 2, 3],
dtype=int32)>

>>> c = tf.Variable(initial_value=[True, False], dtype=tf.bool)
>>> print(c)
<tf.Variable 'Variable:0' shape=(2,) dtype=bool, numpy=array([ True,
False])>

>>> d = tf.Variable(initial_value=['abc'], dtype=tf.string)
>>> print(d)
<tf.Variable 'Variable:0' shape=(1,) dtype=string,
numpy=array([b'abc'], dtype=object)>
```

Notice that we always have to provide the initial values when creating a Variable. Variables have an attribute called trainable, which, by default, is set to True. Higher-level APIs such as Keras will use this attribute to manage the trainable variables and non-trainable ones. You can define a non-trainable Variable as follows:

```
>>> w = tf.Variable([1, 2, 3], trainable=False)
>>> print(w.trainable)
False
```

The values of a Variable can be efficiently modified by running some operations such as .assign(), .assign_add() and related methods. Let's take a look at some examples:

```
>>> print(w.assign([3, 1, 4], read_value=True))
<tf.Variable 'UnreadVariable' shape=(3,) dtype=int32, numpy=array(
[3, 1, 4], dtype=int32)>

>>> w.assign_add([2, -1, 2], read_value=False)
>>> print(w.value())
tf.Tensor([5 0 6], shape=(3,), dtype=int32)
```

When the read_value argument is set to True (which is also the default), these operations will automatically return the new values after updating the current values of the Variable. Setting the read_value to False will suppress the automatic return of the updated value (but the Variable will still be updated in place). Calling w.value() will return the values in a tensor format. Note that we cannot change the shape or type of the Variable during assignment.

You will recall that for NN models, initializing model parameters with random weights is necessary to break the symmetry during backpropagation—otherwise, a multilayer NN would be no more useful than a single-layer NN like logistic regression. When creating a TensorFlow Variable, we can also use a random initialization scheme. TensorFlow can generate random numbers based on a variety of distributions via tf.random (see https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/random). In the following example, we will take a look at some standard initialization methods that are also available in Keras (see https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/initializers).

So, let's look at how we can create a Variable with Glorot initialization, which is a classic random initialization scheme that was proposed by Xavier Glorot and Yoshua Bengio. For this, we create an operator called init as an object of class GlorotNormal. Then, we call this operator and provide the desired shape of the output tensor:

```
>>> tf.random.set_seed(1)
>>> init = tf.keras.initializers.GlorotNormal()
>>> tf.print(init(shape=(3,)))
[-0.722795904 1.01456821 0.251808226]
```

Now, we can use this operator to initialize a `Variable` of shape 2 × 3:

```
>>> v = tf.Variable(init(shape=(2, 3)))
>>> tf.print(v)
[[0.28982234 -0.782292783 -0.0453658961]
 [0.960991383 -0.120003454 0.708528221]]
```

Xavier (or Glorot) initialization

In the early development of deep learning, it was observed that random uniform or random normal weight initialization could often result in a poor performance of the model during training.

In 2010, Glorot and Bengio investigated the effect of initialization and proposed a novel, more robust initialization scheme to facilitate the training of deep networks. The general idea behind Xavier initialization is to roughly balance the variance of the gradients across different layers. Otherwise, some layers may get too much attention during training while the other layers lag behind.

According to the research paper by Glorot and Bengio, if we want to initialize the weights from uniform distribution, we should choose the interval of this uniform distribution as follows:

$$W \sim Uniform\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$$

Here, n_{in} is the number of input neurons that are multiplied by the weights, and n_{out} is the number of output neurons that feed into the next layer. For initializing the weights from Gaussian (normal) distribution, it is recommended that you choose the standard deviation of this Gaussian to be $\sigma = \frac{\sqrt{2}}{\sqrt{n_{in}+n_{out}}}$.

TensorFlow supports Xavier initialization in both uniform and normal distributions of weights.

For more information about Glorot and Bengio's initialization scheme, including the mathematical derivation and proof, read their original paper (*Understanding the difficulty of deep feedforward neural networks*, *Xavier Glorot* and *Yoshua Bengio*, 2010), which is freely available at `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`.

Now, to put this into the context of a more practical use case, let's see how we can define a `Variable` inside the base `tf.Module` class. We will define two variables: a trainable one and a non-trainable one:

```
>>> class MyModule(tf.Module):
...     def __init__(self):
...         init = tf.keras.initializers.GlorotNormal()
...         self.w1 = tf.Variable(init(shape=(2, 3)),
...                               trainable=True)
...         self.w2 = tf.Variable(init(shape=(1, 2)),
...                               trainable=False)
>>> m = MyModule()
>>> print('All module variables:', [v.shape for v in m.variables])
All module variables: [TensorShape([2, 3]), TensorShape([1, 2])]

>>> print('Trainable variable:', [v.shape for v in
...                               m.trainable_variables])
Trainable variable: [TensorShape([2, 3])]
```

As you can see in this code example, subclassing the `tf.Module` class gives us direct access to all variables defined in a given object (here, an instance of our custom `MyModule` class) via the `.variables` attribute.

Finally, let's look at using variables inside a function decorated with `tf.function`. When we define a TensorFlow `Variable` inside a normal function (not decorated), we might expect that a new `Variable` will be created and initialized each time the function is called. However, `tf.function` will try to reuse the `Variable` based on tracing and graph creation. Therefore, TensorFlow does not allow the creation of a `Variable` inside a decorated function and, as a result, the following code will raise an error:

```
>>> @tf.function
... def f(x):
...     w = tf.Variable([1, 2, 3])

>>> f([1])
ValueError: tf.function-decorated function tried to create variables
on non-first call.
```

One way to avoid this problem is to define the `Variable` outside of the decorated function and use it inside the function:

```
>>> w = tf.Variable(tf.random.uniform((3, 3)))
>>> @tf.function
... def compute_z(x):
...     return tf.matmul(w, x)
```

```
>>> x = tf.constant([[1], [2], [3]], dtype=tf.float32)
>>> tf.print(compute_z(x))
```

Computing gradients via automatic differentiation and GradientTape

As you already know, optimizing NNs requires computing the gradients of the cost with respect to the NN weights. This is required for optimization algorithms such as stochastic gradient descent (SGD). In addition, gradients have other applications, such as diagnosing the network to find out why an NN model is making a particular prediction for a test example. Therefore, in this section, we will cover how to compute gradients of a computation with respect to some variables.

Computing the gradients of the loss with respect to trainable variables

TensorFlow supports *automatic differentiation*, which can be thought of as an implementation of the *chain rule* for computing gradients of nested functions. When we define a series of operations that results in some output or even intermediate tensors, TensorFlow provides a context for calculating gradients of these computed tensors with respect to its dependent nodes in the computation graph. In order to compute these gradients, we have to "record" the computations via `tf.GradientTape`.

Let's work with a simple example where we will compute $z = wx + b$ and define the loss as the squared loss between the target and prediction, $Loss = (y - z)^2$. In the more general case, where we may have multiple predictions and targets, we compute the loss as the sum of the squared error, $Loss = \sum_i (y_i - z_i)^2$. In order to implement this computation in TensorFlow, we will define the model parameters, w and b, as variables, and the input, x and y, as tensors. We will place the computation of z and the loss within the `tf.GradientTape` context:

```
>>> w = tf.Variable(1.0)
>>> b = tf.Variable(0.5)
>>> print(w.trainable, b.trainable)
True True

>>> x = tf.convert_to_tensor([1.4])
>>> y = tf.convert_to_tensor([2.1])
>>> with tf.GradientTape() as tape:
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))

>>> dloss_dw = tape.gradient(loss, w)
>>> tf.print('dL/dw:', dloss_dw)
dL/dw: -0.559999764
```

When computing the value z, we could think of the required operations, which we recorded to the "gradient tape," as a forward pass in an NN. We used `tape.gradient` to compute $\frac{\partial Loss}{\partial w}$. Since this is a very simple example, we can obtain the derivatives, $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$, symbolically to verify that the computed gradients match the results we obtained in the previous code example:

```
# verifying the computed gradient
>>> tf.print(2*x*(w*x+b-y))
[-0.559999764]
```

Understanding automatic differentiation

Automatic differentiation represents a set of computational techniques for computing derivatives or gradients of arbitrary arithmetic operations. During this process, gradients of a computation (expressed as a series of operations) are obtained by accumulating the gradients through repeated applications of the chain rule. To better understand the concept behind automatic differentiation, let's consider a series of computations, $y = f\left(g\left(h(x)\right)\right)$, with input x and output y. This can be broken into a series of steps:

- $u_0 = x$
- $u_1 = h(x)$
- $u_2 = g(u_1)$
- $u_3 = f(u_2) = y$

The derivative $\frac{dy}{dx}$ can be computed in two different ways: forward accumulation, which starts with $\frac{du_3}{dx} = \frac{du_3}{du_2}\frac{du_2}{du_0}$, and reverse accumulation, which starts with $\frac{dy}{du_0} = \frac{dy}{du_1}\frac{du_1}{du_0}$ Note that TensorFlow uses the latter, reverse accumulation.

Computing gradients with respect to non-trainable tensors

`tf.GradientTape` automatically supports the gradients for trainable variables. However, for non-trainable variables and other `Tensor` objects, we need to add an additional modification to the `GradientTape` called `tape.watch()` to monitor those as well. For example, if we are interested in computing $\frac{\partial Loss}{\partial x}$, the code will be as follows:

```
>>> with tf.GradientTape() as tape:
...     tape.watch(x)
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))

>>> dloss_dx = tape.gradient(loss, x)

>>> tf.print('dL/dx:', dloss_dx)
dL/dx: [-0.399999857]
```

Adversarial examples

Computing gradients of the loss with respect to the input example is used for generating *adversarial examples* (or *adversarial attacks*). In computer vision, adversarial examples are examples that are generated by adding some small imperceptible noise (or perturbations) to the input example, which results in a deep NN misclassifying them. Covering adversarial examples is beyond the scope of this book, but if you are interested, you can find the original paper by Christian Szegedy et al., titled *Intriguing properties of neural networks,* at `https://arxiv.org/pdf/1312.6199.pdf`.

Keeping resources for multiple gradient computations

When we monitor the computations in the context of `tf.GradientTape`, by default, the tape will keep the resources only for a single gradient computation. For instance, after calling `tape.gradient()` once, the resources will be released and the tape will be cleared. Hence, if we want to compute more than one gradient, for example, both $\frac{\partial Loss}{\partial w}$ and $\frac{\partial Loss}{\partial b}$, we need to make the tape persistent:

```
>>> with tf.GradientTape(persistent=True) as tape:
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))

>>> dloss_dw = tape.gradient(loss, w)
>>> tf.print('dL/dw:', dloss_dw)
dL/dw: -0.559999764

>>> dloss_db = tape.gradient(loss, b)
>>> tf.print('dL/db:', dloss_db)
dL/db: -0.399999857
```

However, keep in mind that this is only needed when we want to compute more than one gradient, as recording and keeping the gradient tape is less memory-efficient compared to releasing the memory after a single gradient computation. This is also why the default setting is `persistent=False`.

Finally, if we are computing gradients of a loss term with respect to the parameters of a model, we can define an optimizer and apply the gradients to optimize the model parameters using the `tf.keras` API, as follows:

```
>>> optimizer = tf.keras.optimizers.SGD()
>>> optimizer.apply_gradients(zip([dloss_dw, dloss_db], [w, b]))

>>> tf.print('Updated w:', w)
Updated w: 1.0056

>>> tf.print('Updated bias:', b)
Updated bias: 0.504
```

You will recall that the initial weight and bias unit were $w = 1.0$ and $b = 0.5$, and applying the gradients of the loss with respect to the model parameters changed the model parameters to $w = 1.0056$ and $b = 0.504$.

Simplifying implementations of common architectures via the Keras API

You have already seen some examples of building a feedforward NN model (for instance, a multilayer perceptron) and defining a sequence of layers using Keras' `Sequential` class. Before we look at different approaches for configuring those layers, let's briefly recap the basic steps by building a model with two densely (fully) connected layers:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=16, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=32, activation='relu'))
>>> ## late variable creation
>>> model.build(input_shape=(None, 4))
>>> model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                multiple                  80
_________________________________________________________________
dense_1 (Dense)              multiple                  544
=================================================================
Total params: 624
Trainable params: 624
Non-trainable params: 0
_________________________________________________________________
```

We specified the input shape with `model.build()`, instantiating the variables after defining the model for that particular shape. The number of parameters of each layer is displayed: 16 × 4 + 16 = 80 for the first layer, and 16 × 32 + 32 = 544 for the second layer. Once variables (or model parameters) are created, we can access both trainable and non-trainable variables as follows:

```
>>> ## printing variables of the model
>>> for v in model.variables:
...     print('{:20s}'.format(v.name), v.trainable, v.shape)
dense/kernel:0       True (4, 16)
dense/bias:0         True (16,)
dense_1/kernel:0     True (16, 32)
dense_1/bias:0       True (32,)
```

In this case, each layer has a weight matrix called `kernel` as well as a bias vector.

Next, let's configure these layers, for example, by applying different activation functions, variable initializers, or regularization methods to the parameters. A comprehensive and complete list of available options for these categories can be found in the official documentation:

- Choosing activation functions via `tf.keras.activations`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/activations`
- Initializing the layer parameters via `tf.keras.initializers`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/initializers`
- Applying regularization to the layer parameters (to prevent overfitting) via `tf.keras.regularizers`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/regularizers`

In the following code example, we will configure the first layer by specifying initializers for the kernel and bias variables. Then, we will configure the second layer by specifying an L1 regularizer for the kernel (weight matrix):

```
>>> model = tf.keras.Sequential()
>>> model.add(
...     tf.keras.layers.Dense(
...         units=16,
...         activation=tf.keras.activations.relu,
...         kernel_initializer= \
...             tf.keras.initializers.glorot_uniform(),
...         bias_initializer=tf.keras.initializers.Constant(2.0)
...     ))

>>> model.add(
...     tf.keras.layers.Dense(
...         units=32,
...         activation=tf.keras.activations.sigmoid,
...         kernel_regularizer=tf.keras.regularizers.l1
...     ))
```

Furthermore, in addition to configuring the individual layers, we can also configure the model when we compile it. We can specify the type of optimizer and the loss function for training, as well as which metrics to use for reporting the performance on the training, validation, and test datasets. Again, a comprehensive list of all available options can be found in the official documentation:

- Optimizers via `tf.keras.optimizers`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/optimizers`
- Loss functions via `tf.keras.losses`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/losses`
- Performance metrics via `tf.keras.metrics`: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/metrics`

Choosing a loss function

Regarding the choices for optimization algorithms, SGD and Adam are the most widely used methods. The choice of loss function depends on the task; for example, you might use mean square error loss for a regression problem.

The family of cross-entropy loss functions supplies the possible choices for classification tasks, which are extensively discussed in *Chapter 15, Classifying Images with Deep Convolutional Neural Networks*.

Furthermore, you can use the techniques you have learned from previous chapters (for example, techniques for model evaluation from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*) combined with the appropriate metrics for the problem. For example, precision and recall, accuracy, area under the curve (AUC), and false negative and false positive scores are appropriate metrics for evaluating classification models.

In this example, we will compile the model using the SGD optimizer, cross-entropy loss for binary classification, and a specific list of metrics, including accuracy, precision, and recall:

```
>>> model.compile(
...     optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
...     loss=tf.keras.losses.BinaryCrossentropy(),
...     metrics=[tf.keras.metrics.Accuracy(),
...              tf.keras.metrics.Precision(),
...              tf.keras.metrics.Recall(),])
```

When we train this model by calling `model.fit(...)`, the history of the loss and the specified metrics for evaluating training and validation performance (if a validation dataset is used) will be returned, which can be used to diagnose the learning behavior.

Next, we will look at a more practical example: solving the classic XOR classification problem using the Keras API. First, we will use the `tf.keras.Sequential()` class to build the model. Along the way, you will also learn about the capacity of a model for handling nonlinear decision boundaries. Then, we will cover other ways of building a model that will give us more flexibility and control over the layers of the network.

Solving an XOR classification problem

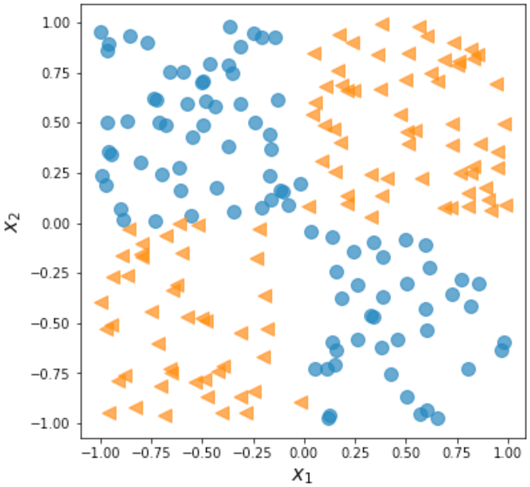
The XOR classification problem is a classic problem for analyzing the capacity of a model with regard to capturing the nonlinear decision boundary between two classes. We generate a toy dataset of 200 training examples with two features (x_0, x_1) drawn from a uniform distribution between $[-1, 1)$. Then, we assign the ground truth label for training example *i* according to the following rule:

$$y^{(i)} = \begin{cases} 0 & \text{if } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{otherwise} \end{cases}$$

We will use half of the data (100 training examples) for training and the remaining half for validation. The code for generating the data and splitting it into the training and validation datasets is as follows:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1]<0] = 0

>>> x_train = x[:100, :]
>>> y_train = y[:100]
>>> x_valid = x[100:, :]
>>> y_valid = y[100:]

>>> fig = plt.figure(figsize=(6, 6))
>>> plt.plot(x[y==0, 0],
...          x[y==0, 1], 'o', alpha=0.75, markersize=10)
>>> plt.plot(x[y==1, 0],
...          x[y==1, 1], '<', alpha=0.75, markersize=10)
>>> plt.xlabel(r'$x_1$', size=15)
>>> plt.ylabel(r'$x_2$', size=15)
>>> plt.show()
```

The code results in the following scatterplot of the training and validation examples, shown with different markers based on their class label:



In the previous subsection, we covered the essential tools that we need to implement a classifier in TensorFlow. We now need to decide what architecture we should choose for this task and dataset. As a general rule of thumb, the more layers we have, and the more neurons we have in each layer, the larger the capacity of the model will be. Here, the model capacity can be thought of as a measure of how readily the model can approximate complex functions. While having more parameters means the network can fit more complex functions, larger models are usually harder to train (and prone to overfitting). In practice, it is always a good idea to start with a simple model as a base line, for example, a single-layer NN like logistic regression:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=1,
...                                  input_shape=(2,),
...                                  activation='sigmoid'))
>>> model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 3
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

The total size of the parameters for this simple logistic regression model is 3: a weight matrix (or kernel) of size 2 × 1 and a bias vector of size 1. After defining the model, we will compile the model and train it for 200 epochs using a batch size of 2:

```
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> hist = model.fit(x_train, y_train,
...                  validation_data=(x_valid, y_valid),
...                  epochs=200, batch_size=2, verbose=0)
```

Notice that `model.fit()` returns a history of training epochs, which is useful for visual inspection after training. In the following code, we will plot the learning curves, including the training and validation loss, as well as their accuracies.

We will also use the MLxtend library to visualize the validation data and the decision boundary.

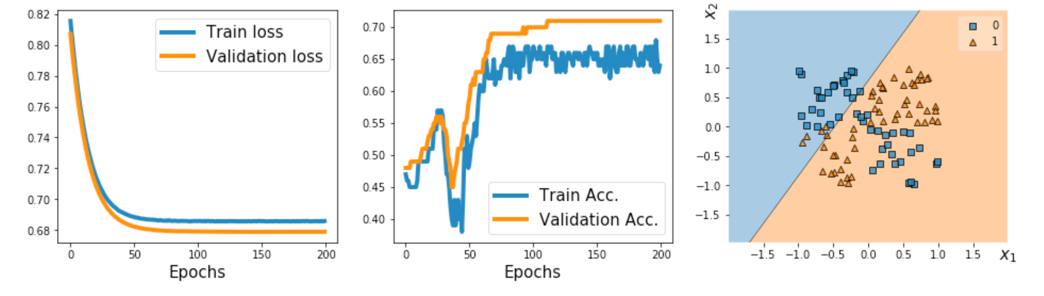
MLxtend can be installed via `conda` or `pip` as follows:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

The following code will plot the training performance along with the decision region bias:

```
>>> from mlxtend.plotting import plot_decision_regions

>>> history = hist.history

>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history['loss'], lw=4)
>>> plt.plot(history['val_loss'], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)

>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history['binary_accuracy'], lw=4)
>>> plt.plot(history['val_binary_accuracy'], lw=4)
>>> plt.legend(['Train Acc.', 'Validation Acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)

>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid, y=y_valid.astype(np.integer),
...                       clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

This results in the following figure, with three separate panels for the losses, accuracies, and the scatterplot of the validation examples, along with the decision boundary:



As you can see, a simple model with no hidden layer can only derive a linear decision boundary, which is unable to solve the XOR problem. As a consequence, we can observe that the loss terms for both the training and the validation datasets are very high, and the classification accuracy is very low.

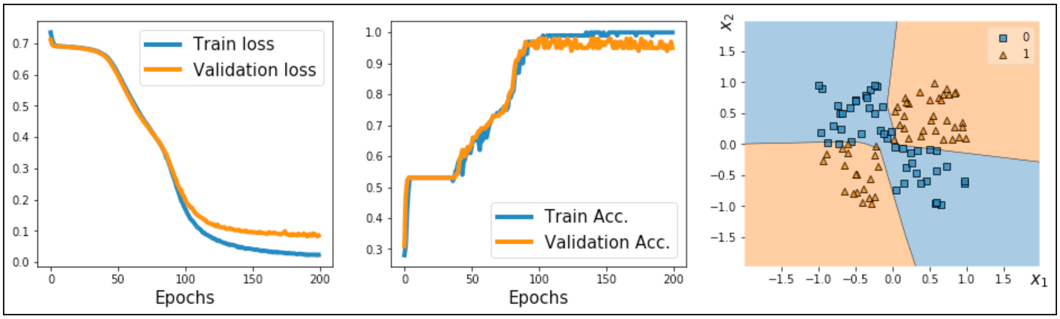
In order to derive a nonlinear decision boundary, we can add one or more hidden layers connected via nonlinear activation functions. The universal approximation theorem states that a feedforward NN with a single hidden layer and a relatively large number of hidden units can approximate arbitrary continuous functions relatively well. Thus, one approach for tackling the XOR problem more satisfactorily is to add a hidden layer and compare different numbers of hidden units until we observe satisfactory results on the validation dataset. Adding more hidden units would correspond to increasing the width of a layer.

Alternatively, we can also add more hidden layers, which will make the model deeper. The advantage of making a network deeper rather than wider is that fewer parameters are required to achieve a comparable model capacity. However, a downside of deep (versus wide) models is that deep models are prone to vanishing and exploding gradients, which make them harder to train.

As an exercise, try adding one, two, three, and four hidden layers, each with four hidden units. In the following example, we will take a look at the results of a feedforward NN with three hidden layers:

```
>>> tf.random.set_seed(1)
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=4, input_shape=(2,),
...                                 activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=4, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=4, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

>>> model.summary()
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 4)                 12
_________________________________________________________________
dense_12 (Dense)             (None, 4)                 20
_________________________________________________________________
dense_13 (Dense)             (None, 4)                 20
_________________________________________________________________
dense_14 (Dense)             (None, 1)                 5
=================================================================
Total params: 57
Trainable params: 57
Non-trainable params: 0
_________________________________________________________________

>>> ## compile:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> ## train:
>>> hist = model.fit(x_train, y_train,
...                  validation_data=(x_valid, y_valid),
...                  epochs=200, batch_size=2, verbose=0)
```

We can repeat the previous code for visualization, which produces the following:



Now, we can see that the model is able to derive a nonlinear decision boundary for this data, and the model reaches 100 percent accuracy on the training dataset. The validation dataset's accuracy is 95 percent, which indicates that the model is slightly overfitting.

Making model building more flexible with Keras' functional API

In the previous example, we used the Keras `Sequential` class to create a fully connected NN with multiple layers. This is a very common and convenient way of building models. However, it unfortunately doesn't allow us to create more complex models that have multiple input, output, or intermediate branches. That's where Keras' so-called functional API comes in handy.

To illustrate how the functional API can be used, we will implement the same architecture that we built using the objected-oriented (`Sequential`) approach in the previous section; however, this time, we will use the functional approach. In this approach, we first specify the input. Then, the hidden layers are constructed, with their outputs named `h1`, `h2`, and `h3`. For this problem, we use the output of each layer as the input to the succedent layer (note that if you are building more complex models that have multiple branches, this may not be the case, but it can still be done via the functional API). Finally, we specify the output as the final dense layer that receives `h3` as input. The code for this is as follows:

```
>>> tf.random.set_seed(1)

>>> ## input layer:
>>> inputs = tf.keras.Input(shape=(2,))

>>> ## hidden layers
>>> h1 = tf.keras.layers.Dense(units=4, activation='relu')(inputs)
>>> h2 = tf.keras.layers.Dense(units=4, activation='relu')(h1)
>>> h3 = tf.keras.layers.Dense(units=4, activation='relu')(h2)

>>> ## output:
>>> outputs = tf.keras.layers.Dense(units=1, activation='sigmoid')(h3)

>>> ## construct a model:
>>> model = tf.keras.Model(inputs=inputs, outputs=outputs)

>>> model.summary()
```

Compiling and training this model is similar to what we did previously:

```
>>> ## compile:
>>> model.compile(
...     optimizer=tf.keras.optimizers.SGD(),
...     loss=tf.keras.losses.BinaryCrossentropy(),
...     metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> ## train:
>>> hist = model.fit(
...     x_train, y_train,
...     validation_data=(x_valid, y_valid),
...     epochs=200, batch_size=2, verbose=0)
```

Implementing models based on Keras' Model class

An alternative way to build complex models is by subclassing `tf.keras.Model`. In this approach, we create a new class derived from `tf.keras.Model` and define the function, `__init__()`, as a constructor. The `call()` method is used to specify the forward pass. In the constructor function, `__init__()`, we define the layers as attributes of the class so that they can be accessed via the `self` reference attribute. Then, in the `call()` method, we specify how these layers are to be used in the forward pass of the NN. The code for defining a new class that implements the previous model is as follows:

```
>>> class MyModel(tf.keras.Model):
...     def __init__(self):
...         super(MyModel, self).__init__()
...         self.hidden_1 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.hidden_2 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.hidden_3 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.output_layer = tf.keras.layers.Dense(
...             units=1, activation='sigmoid')
...
...     def call(self, inputs):
...         h = self.hidden_1(inputs)
...         h = self.hidden_2(h)
...         h = self.hidden_3(h)
...         return self.output_layer(h)
```

Notice that we used the same output name, `h`, for all hidden layers. This makes the code more readable and easier to follow.

A model class derived from `tf.keras.Model` through subclassing inherits general model attributes, such as `build()`, `compile()`, and `fit()`. Therefore, once we define an instance of this new class, we can compile and train it like any other model built by Keras:

```
>>> tf.random.set_seed(1)

>>> model = MyModel()
>>> model.build(input_shape=(None, 2))

>>> model.summary()
>>> ## compile:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> ## train:
>>> hist = model.fit(x_train, y_train,
...                  validation_data=(x_valid, y_valid),
...                  epochs=200, batch_size=2, verbose=0)
```

Writing custom Keras layers

In cases where we want to define a new layer that is not already supported by Keras, we can define a new class derived from the `tf.keras.layers.Layer` class. This is especially useful when designing a new layer or customizing an existing layer.

To illustrate the concept of implementing custom layers, let's consider a simple example. Imagine we want to define a new linear layer that computes $w(x + \epsilon) + b$, where ϵ refers to a random variable as a noise variable. To implement this computation, we define a new class as a subclass of `tf.keras.layers.Layer`. For this new class, we have to define both the constructor `__init__()` method and the `call()` method. In the constructor, we define the variables and other required tensors for our customized layer. We have the option to create variables and initialize them in the constructor if the `input_shape` is given to the constructor. Alternatively, we can delay the variable initialization (for instance, if we do not know the exact input shape upfront) and delegate it to the `build()` method for late variable creation. In addition, we can define `get_config()` for serialization, which means that a model using our custom layer can be efficiently saved using TensorFlow's model saving and loading capabilities.

To look at a concrete example, we are going to define a new layer called `NoisyLinear`, which implements the computation $w(x + \epsilon) + b$, which was mentioned in the preceding paragraph:

```
>>> class NoisyLinear(tf.keras.layers.Layer):
...     def __init__(self, output_dim, noise_stddev=0.1, **kwargs):
...         self.output_dim = output_dim
...         self.noise_stddev = noise_stddev
...         super(NoisyLinear, self).__init__(**kwargs)
...
...     def build(self, input_shape):
...         self.w = self.add_weight(name='weights',
...                                  shape=(input_shape[1],
...                                         self.output_dim),
...                                  initializer='random_normal',
...                                  trainable=True)
...
...         self.b = self.add_weight(shape=(self.output_dim,),
...                                  initializer='zeros',
...                                  trainable=True)
...
...     def call(self, inputs, training=False):
...         if training:
...             batch = tf.shape(inputs)[0]
...             dim = tf.shape(inputs)[1]
...             noise = tf.random.normal(shape=(batch, dim),
...                                      mean=0.0,
...                                      stddev=self.noise_stddev)
...
...             noisy_inputs = tf.add(inputs, noise)
...         else:
...             noisy_inputs = inputs
...         z = tf.matmul(noisy_inputs, self.w) + self.b
...         return tf.keras.activations.relu(z)
...
...     def get_config(self):
...         config = super(NoisyLinear, self).get_config()
...         config.update({'output_dim': self.output_dim,
...                        'noise_stddev': self.noise_stddev})
...         return config
```

In the constructor, we have added an argument, noise_stddev, to specify the standard deviation for the distribution of ϵ, which is sampled from a Gaussian distribution. Furthermore, notice that in the call() method, we have used an additional argument, training=False. In the context of Keras, the training argument is a special Boolean argument that distinguishes whether a model or layer is used during training (for example, via fit()) or only for prediction (for example, via predict(); this is sometimes also called "inference" or evaluation). One of the main differences between training and prediction is that during prediction, we do not require gradients. Also, there are certain methods that behave differently in training and prediction modes. You will encounter an example of such a method, Dropout, in the upcoming chapters. In the previous code snippet, we also specified that the random vector, ϵ, was to be generated and added to the input during training only and not used for inference or evaluation.

Before we go a step further and use our custom NoisyLinear layer in a model, let's test it in the context of a simple example.

In the following code, we will define a new instance of this layer, initialize it by calling `.build()`, and execute it on an input tensor. Then, we will serialize it via `.get_config()` and restore the serialized object via `.from_config()`:

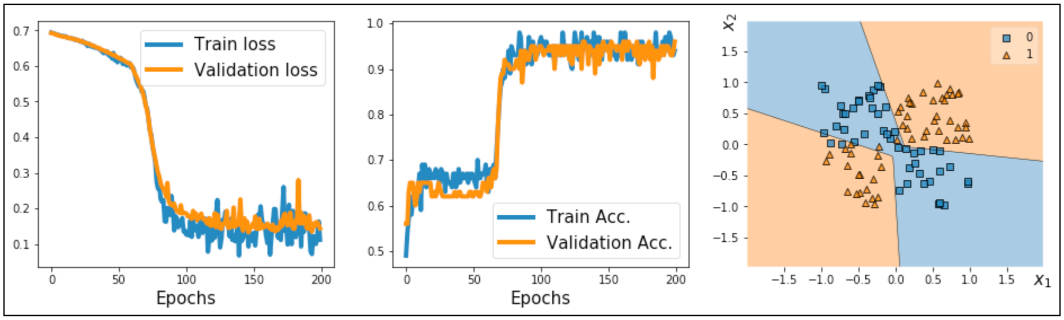
```
>>> tf.random.set_seed(1)
>>> noisy_layer = NoisyLinear(4)
>>> noisy_layer.build(input_shape=(None, 4))

>>> x = tf.zeros(shape=(1, 4))
>>> tf.print(noisy_layer(x, training=True))
[[0 0.00821428 0 0]]

>>> ## re-building from config:
>>> config = noisy_layer.get_config()
>>> new_layer = NoisyLinear.from_config(config)
>>> tf.print(new_layer(x, training=True))
[[0 0.0108502861 0 0]]
```

In the previous code snippet, we called the layer two times on the same input tensor. However, note that the outputs differ because the `NoisyLinear` layer added random noise to the input tensor.

Now, let's create a new model similar to the previous one for solving the XOR classification task. As before, we will use Keras' `Sequential` class, but this time, we will use our `NoisyLinear` layer as the first hidden layer of the multilayer perceptron. The code is as follows:

```
>>> tf.random.set_seed(1)

>>> model = tf.keras.Sequential([
...     NoisyLinear(4, noise_stddev=0.1),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=1, activation='sigmoid')])

>>> model.build(input_shape=(None, 2))
>>> model.summary()

>>> ## compile:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> ## train:
>>> hist = model.fit(x_train, y_train,
...                  validation_data=(x_valid, y_valid),
...                  epochs=200, batch_size=2,
...                  verbose=0)

>>> ## Plotting
>>> history = hist.history

>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history['loss'], lw=4)
>>> plt.plot(history['val_loss'], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)

>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history['binary_accuracy'], lw=4)
>>> plt.plot(history['val_binary_accuracy'], lw=4)
>>> plt.legend(['Train Acc.', 'Validation Acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)

>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid, y=y_valid.astype(np.integer),
...                       clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

The resulting figure will be as follows:



Here, our goal was to learn how to define a new custom layer subclassed from `tf.keras.layers.Layer` and to use it as we would use any other standard Keras layer. Although, with this particular example, `NoisyLinear` did not help to improve the performance, please keep in mind that our objective was to mainly learn how to write a customized layer from scratch. In general, writing a new customized layer can be useful in other applications, for example, if you develop a new algorithm that depends on a new layer beyond the existing ones.

TensorFlow Estimators

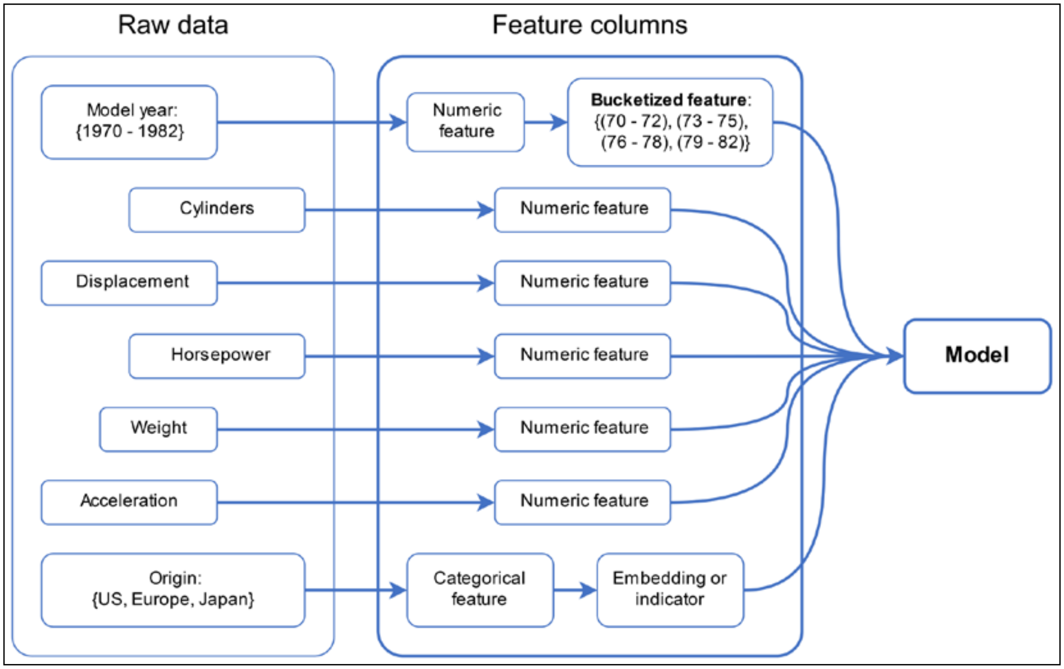
So far, in this chapter, we have mostly focused on the low-level TensorFlow API. We used decorators to modify functions to compile the computational graphs explicitly for computational efficiency. Then, we worked with the Keras API and implemented feedforward NNs, to which we added customized layers. In this section, we will switch gears and work with TensorFlow Estimators. The `tf.estimator` API encapsulates the underlying steps in machine learning tasks, such as training, prediction (inference), and evaluation. Estimators are more encapsulated but also more scalable when compared to the previous approaches that we have covered in this chapter. Also, the `tf.estimator` API adds support for running models on multiple platforms without requiring major code changes, which makes them more suitable for the so-called "production phase" in industry applications. In addition, TensorFlow comes with a selection of off-the-shelf estimators for common machine learning and deep learning architectures that are useful for comparison studies, for example, to quickly assess whether a certain approach is applicable to a particular dataset or problem.

In the remaining sections of this chapter, you will learn how to use such pre-made Estimators and how to create an Estimator from an existing Keras model. One of the essential elements of Estimators is defining the feature columns as a mechanism for importing data into an Estimator-based model, which we will cover in the next section.

Working with feature columns

In machine learning and deep learning applications, we can encounter various different types of features: continuous, unordered categorical (nominal), and ordered categorical (ordinal). You will recall that in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, we covered different types of features and learned how to handle each type. Note that while numeric data can be either continuous or discrete, in the context of the TensorFlow API, "numeric" data specifically refers to continuous data of the floating point type.

Sometimes, feature sets are comprised of a mixture of different feature types. While TensorFlow Estimators were designed to handle all these different types of features, we must specify how each feature should be interpreted by the Estimator. For example, consider a scenario with a set of seven different features, as shown in the following figure:



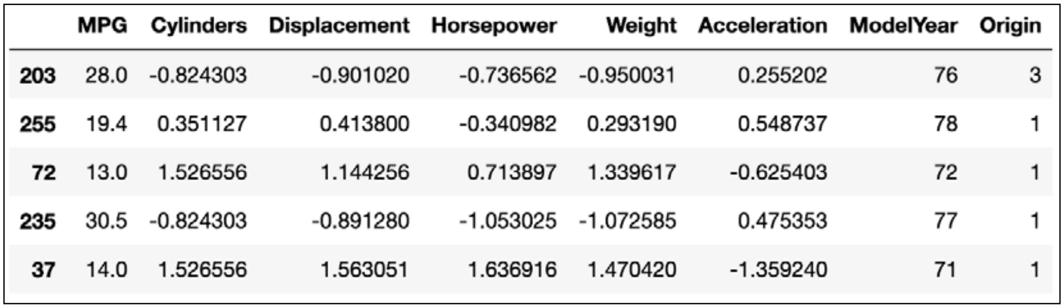
The features shown in the figure (model year, cylinders, displacement, horsepower, weight, acceleration, and origin) were obtained from the Auto MPG dataset, which is a common machine learning benchmark dataset for predicting the fuel efficiency of a car in miles per gallon (MPG). The full dataset and its description are available from UCI's machine learning repository at `https://archive.ics.uci.edu/ml/datasets/auto+mpg`.

We are going to treat five features from the Auto MPG dataset (number of cylinders, displacement, horsepower, weight, and acceleration) as "numeric" (here, continuous) features. The model year can be regarded as an ordered categorical (ordinal) feature. Lastly, the manufacturing origin can be regarded as an unordered categorical (nominal) feature with three possible discrete values, 1, 2, and 3, which correspond to the US, Europe, and Japan, respectively.

Let's first load the data and apply the necessary preprocessing steps, such as partitioning the dataset into training and test datasets, as well as standardizing the continuous features:

```
>>> import pandas as pd

>>> dataset_path = tf.keras.utils.get_file(
...     "auto-mpg.data",
...     ("http://archive.ics.uci.edu/ml/machine-learning"
...      "-databases/auto-mpg/auto-mpg.data"))

>>> column_names = [
...     'MPG', 'Cylinders', 'Displacement',
...     'Horsepower', 'Weight', 'Acceleration',
...     'ModelYear', 'Origin']

>>> df = pd.read_csv(dataset_path, names=column_names,
...                  na_values = '?', comment='\t',
...                  sep=' ', skipinitialspace=True)
>>> ## drop the NA rows
>>> df = df.dropna()
>>> df = df.reset_index(drop=True)

>>> ## train/test splits:
>>> import sklearn
>>> import sklearn.model_selection

>>> df_train, df_test = sklearn.model_selection.train_test_split(
...    df, train_size=0.8)
>>> train_stats = df_train.describe().transpose()

>>> numeric_column_names = [
...     'Cylinders', 'Displacement',
...     'Horsepower', 'Weight',
...     'Acceleration']

>>> df_train_norm, df_test_norm = df_train.copy(), df_test.copy()

>>> for col_name in numeric_column_names:
...     mean = train_stats.loc[col_name, 'mean']
...     std  = train_stats.loc[col_name, 'std']
...     df_train_norm.loc[:, col_name] = (
...         df_train_norm.loc[:, col_name] - mean)/std
...     df_test_norm.loc[:, col_name] = (
...         df_test_norm.loc[:, col_name] - mean)/std

>>> df_train_norm.tail()
```

This results in the following:



	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	ModelYear	Origin
203	28.0	-0.824303	-0.901020	-0.736562	-0.950031	0.255202	76	3
255	19.4	0.351127	0.413800	-0.340982	0.293190	0.548737	78	1
72	13.0	1.526556	1.144256	0.713897	1.339617	-0.625403	72	1
235	30.5	-0.824303	-0.891280	-1.053025	-1.072585	0.475353	77	1
37	14.0	1.526556	1.563051	1.636916	1.470420	-1.359240	71	1

The pandas `DataFrame` that we created via the previous code snippet contains five columns with values of the type `float`. These columns will constitute the continuous features. In the following code, we will use TensorFlow's `feature_column` function to transform these continuous features into the feature column data structure that TensorFlow Estimators can work with:

```
>>> numeric_features = []
>>> for col_name in numeric_column_names:
...     numeric_features.append(
...         tf.feature_column.numeric_column(key=col_name))
```

Next, let's group the rather fine-grained model year information into buckets to simplify the learning task for the model that we are going to train later. Concretely, we are going to assign each car into one of four "year" buckets, as follows:

$$\text{bucket} = \begin{cases} 0 & \text{if year} < 73 \\ 1 & \text{if } 73 \leq \text{year} \leq 76 \\ 2 & \text{if } 76 \leq \text{year} \leq 79 \\ 3 & \text{if year} \geq 79 \end{cases}$$

Note that the chosen intervals were selected arbitrarily to illustrate the concepts of "bucketing." In order to group the cars into these buckets, we will first define a numeric feature based on each original model year. Then, these numeric features will be passed to the `bucketized_column` function for which we will specify three interval cut-off values: [73, 76, 79]. The specified values include the right cut-off value. These cut-off values are used to specify half-closed intervals, for instance, $(-\infty, 73)$, $[73, 76)$, $[76, 79)$, and $[79, \infty)$. The code is as follows:

```
>>> feature_year = tf.feature_column.numeric_column(key='ModelYear')
>>> bucketized_features = []
>>> bucketized_features.append(
...     tf.feature_column.bucketized_column(
...         source_column=feature_year,
...         boundaries=[73, 76, 79]))
```

For consistency, we added this bucketized feature column to a Python list, even though the list consists of only one entry. In the following steps, we will merge this list with the lists made from other features, which will then be provided as input to the TensorFlow Estimator-based model.

Next, we will proceed with defining a list for the unordered categorical feature, Origin. In TensorFlow, there are different ways of creating a categorical feature column. If the data contains the category names (for example, in string format like "US," "Europe," and "Japan"), then we can use tf.feature_column.categorical_column_with_vocabulary_list and provide a list of unique, possible category names as input. If the list of possible categories is too large, for example, in a typical text analysis context, then we can use tf.feature_column.categorical_column_with_vocabulary_file instead. When using this function, we simply provide a file that contains all the categories/words so that we do not have to store a list of all possible words in memory. Moreover, if the features are already associated with an index of categories in the range [0, num_categories), then we can use the tf.feature_column.categorical_column_with_identity function. However, in this case, the feature Origin is given as integer values 1, 2, 3 (as opposed to 0, 1, 2), which does not match the requirement for categorical indexing, as it expects the indices to start from 0.

In the following code example, we will proceed with the vocabulary list:

```
>>> feature_origin = tf.feature_column.categorical_column_with_
vocabulary_list(
...     key='Origin',
...     vocabulary_list=[1, 2, 3])
```

Certain Estimators, such as DNNClassifier and DNNRegressor, only accept so-called "dense columns." Therefore, the next step is to convert the existing categorical feature column to such a dense column. There are two ways to do this: using an embedding column via embedding_column or an indicator column via indicator_column. An indicator column converts the categorical indices to one-hot encoded vectors, for example, index 0 will be encoded as [1, 0, 0], index 1 will be encoded as [0, 1, 0], and so on. On the other hand, the embedding column maps each index to a vector of random number of the type float, which can be trained.

When the number of categories is large, using the embedding column with fewer dimensions than the number of categories can improve the performance. In the following code snippet, we will use the indicator column approach on the categorical feature in order to convert it into the dense format:

```
>>> categorical_indicator_features = []
>>> categorical_indicator_features.append(
...     tf.feature_column.indicator_column(feature_origin))
```

In this section, we have covered the most common approaches for creating feature columns that can be used with TensorFlow Estimators. However, there are several additional feature columns that we haven't discussed, including hashed columns and crossed columns. More information about these other feature columns can be found in the official TensorFlow documentation at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column`.

Machine learning with pre-made Estimators

Now, after constructing the mandatory feature columns, we can finally utilize TensorFlow's Estimators. Using pre-made Estimators can be summarized in four steps:

1. Define an input function for data loading
2. Convert the dataset into feature columns
3. Instantiate an Estimator (use a pre-made Estimator or create a new one, for example, by converting a Keras model into an Estimator)
4. Use the Estimator methods `train()`, `evaluate()`, and `predict()`

Continuing with the Auto MPG example from the previous section, we will apply these four steps to illustrate how we can use Estimators in practice. For the first step, we need to define a function that processes the data and returns a TensorFlow dataset consisting of a tuple that contains the input features and the labels (ground truth MPG values). Note that the features must be in a dictionary format, and the keys of the dictionary must match the feature columns' names.

Starting with the first step, we will define the input function for the training data as follows:

```
>>> def train_input_fn(df_train, batch_size=8):
...     df = df_train.copy()
...     train_x, train_y = df, df.pop('MPG')
...     dataset = tf.data.Dataset.from_tensor_slices(
...         (dict(train_x), train_y))
...
```

```
...     # shuffle, repeat, and batch the examples.
...     return dataset.shuffle(1000).repeat().batch(batch_size)
```

Notice that we used `dict(train_x)` in this function to convert the pandas `DataFrame` object into a Python dictionary. Let's load a batch from this dataset to see how it looks:

```
>>> ds = train_input_fn(df_train_norm)
>>> batch = next(iter(ds))
>>> print('Keys:', batch[0].keys())
Keys: dict_keys(['Cylinders', 'Displacement', 'Horsepower', 'Weight',
'Acceleration', 'ModelYear', 'Origin'])

>>> print('Batch Model Years:', batch[0]['ModelYear'])
Batch Model Years: tf.Tensor([74 71 81 72 82 81 70 74], shape=(8,),
dtype=int32)
```

We also need to define an input function for the test dataset that will be used for evaluation after model training:

```
>>> def eval_input_fn(df_test, batch_size=8):
...     df = df_test.copy()
...     test_x, test_y = df, df.pop('MPG')
...     dataset = tf.data.Dataset.from_tensor_slices(
...         (dict(test_x), test_y))
...     return dataset.batch(batch_size)
```

Now, moving on to step 2, we need to define the feature columns. We have already defined a list containing the continuous features, a list for the bucketized feature column, and a list for the categorical feature column. We can now concatenate these individual lists to a single list containing all feature columns:

```
>>> all_feature_columns = (
...     numeric_features +
...     bucketized_features +
...     categorical_indicator_features)
```

For step 3, we need to instantiate a new Estimator. Since predicting MPG values is a typical regression problem, we will use `tf.estimator.DNNRegressor`. When instantiating the regression Estimator, we will provide the list of feature columns and specify the number of hidden units that we want to have in each hidden layer using the argument `hidden_units`. Here, we will use two hidden layers, where the first hidden layer has 32 units and the second hidden layer has 10 units:

```
>>> regressor = tf.estimator.DNNRegressor(
...     feature_columns=all_feature_columns,
...     hidden_units=[32, 10],
...     model_dir='models/autompg-dnnregressor/')
```

The other argument, model_dir, that we have provided specifies the directory for saving model parameters. One of the advantages of Estimators is that they automatically checkpoint the model during training, so that in case the training of the model crashes for an unexpected reason (like power failure), we can easily load the last saved checkpoint and continue training from there. The checkpoints will also be saved in the directory specified by model_dir. If we do not specify the model_dir argument, the Estimator will create a random temporary folder (for example, in the Linux operating system, a random folder in the /tmp/ directory will be created), which will be used for this purpose.

After these three basic setup steps, we can finally use the Estimator for training, evaluation, and, eventually, prediction. The regressor can be trained by calling the train() method, for which we require the previously defined input function:

```
>>> EPOCHS = 1000
>>> BATCH_SIZE = 8
>>> total_steps = EPOCHS * int(np.ceil(len(df_train) / BATCH_SIZE))
>>> print('Training Steps:', total_steps)
Training Steps: 40000

>>> regressor.train(
...     input_fn=lambda:train_input_fn(
...         df_train_norm, batch_size=BATCH_SIZE),
...     steps=total_steps)
```

Calling .train() will automatically save the checkpoints during the training of the model. We can then reload the last checkpoint:

```
>>> reloaded_regressor = tf.estimator.DNNRegressor(
...     feature_columns=all_feature_columns,
...     hidden_units=[32, 10],
...     warm_start_from='models/autompg-dnnregressor/',
...     model_dir='models/autompg-dnnregressor/')
```

Then, in order to evaluate the predictive performance of the trained model, we can use the evaluate() method, as follows:

```
>>> eval_results = reloaded_regressor.evaluate(
...     input_fn=lambda:eval_input_fn(df_test_norm, batch_size=8))
>>> print('Average-Loss {:.4f}'.format(
...      eval_results['average_loss']))
Average-Loss 15.1866
```

Finally, to predict the target values on new data points, we can use the predict() method. For the purposes of this example, suppose that the test dataset represents a dataset of new, unlabeled data points in a real-world application.

Note that in a real-world prediction task, the input function will only need to return a dataset consisting of features, assuming that the labels are not available. Here, we will simply use the same input function that we used for evaluation to get the predictions for each example:

```
>>> pred_res = regressor.predict(
...     input_fn=lambda: eval_input_fn(
...         df_test_norm, batch_size=8))
>>> print(next(iter(pred_res)))
{'predictions': array([23.747658], dtype=float32)}
```

While the preceding code snippets conclude the illustration of the four steps that are required for using pre-made Estimators, for practice, let's take a look at another pre-made Estimator: the boosted tree regressor, `tf.estimator.BoostedTreeRegressor`. Since, the input functions and the feature columns are already built, we just need to repeat steps 3 and 4. For step 3, we will create an instance of `BoostedTreeRegressor` and configure it to have 200 trees.

Decision tree boosting

We already covered the ensemble algorithms, including boosting, in *Chapter 7, Combining Different Models for Ensemble Learning*. The boosted tree algorithm is a special family of boosting algorithms that is based on the optimization of an arbitrary loss function. Feel free to visit `https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d` to learn more.

```
>>> boosted_tree = tf.estimator.BoostedTreesRegressor(
...     feature_columns=all_feature_columns,
...     n_batches_per_layer=20,
...     n_trees=200)

>>> boosted_tree.train(
...     input_fn=lambda:train_input_fn(
...         df_train_norm, batch_size=BATCH_SIZE))

>>> eval_results = boosted_tree.evaluate(
...     input_fn=lambda:eval_input_fn(
...         df_test_norm, batch_size=8))

>>> print('Average-Loss {:.4f}'.format(
...       eval_results['average_loss']))
Average-Loss 11.2609
```

As you can see, the boosted tree regressor achieves lower average loss than the `DNNRegressor`. For a small dataset like this, this is expected.

In this section, we covered the essential steps for using TensorFlow's Estimators for regression. In the next subsection, we will take a look at a typical classification example using Estimators.

Using Estimators for MNIST handwritten digit classification

For this classification problem, we are going to use the `DNNClassifier` Estimator provided by TensorFlow, which lets us implement a multilayer perceptron very conveniently. In the previous section, we covered the four essential steps for using the pre-made Estimators in detail, which we will need to repeat in this section. First, we are going to import the `tensorflow_datasets` (`tfds`) submodule, which we can use to load the MNIST dataset and specify the hyperparameters of the model.

Estimator API and graph issues

Since parts of TensorFlow 2.0 are still a bit rough around the edges, you may encounter the following issue when executing the next code block: `RuntimeError: Graph is finalized and cannot be modified.` Currently, there is no good solution for this issue, and a suggested workaround is to restart your Python, IPython, or Jupyter Notebook session before executing the next code block.

The setup step includes loading the dataset and specifying hyperparameters (`BUFFER_SIZE` for shuffling the dataset, `BATCH_SIZE` for the size of mini-batches, and the number of training epochs):

```
>>> import tensorflow_datasets as tfds
>>> import tensorflow as tf
>>> import numpy as np

>>> BUFFER_SIZE = 10000
>>> BATCH_SIZE = 64
>>> NUM_EPOCHS = 20
>>> steps_per_epoch = np.ceil(60000 / BATCH_SIZE)
```

Note that `steps_per_epoch` determines the number of iterations in each epoch, which is needed for infinitely repeated datasets (as discussed in *Chapter 13, Parallelizing Neural Network Training with TensorFlow*. Next, we will define a helper function that will preprocess the input image and its label.

Since the input image is originally of the type 'uint8' (in the range [0, 255]), we will use tf.image.convert_image_dtype() to convert its type to tf.float32 (and thereby, within the range [0, 1]):

```
>>> def preprocess(item):
...     image = item['image']
...     label = item['label']
...     image = tf.image.convert_image_dtype(
...         image, tf.float32)
...     image = tf.reshape(image, (-1,))
...
...     return {'image-pixels':image}, label[..., tf.newaxis]
```

Step 1: Define two input functions (one for training and one for evaluation):

```
>>> ## Step 1: Define the input functions
>>> def train_input_fn():
...     datasets = tfds.load(name='mnist')
...     mnist_train = datasets['train']
...
...     dataset = mnist_train.map(preprocess)
...     dataset = dataset.shuffle(BUFFER_SIZE)
...     dataset = dataset.batch(BATCH_SIZE)
...     return dataset.repeat()

>>> def eval_input_fn():
...     datasets = tfds.load(name='mnist')
...     mnist_test = datasets['test']
...     dataset = mnist_test.map(preprocess).batch(BATCH_SIZE)
...     return dataset
```

Notice that the dictionary of features has only one key, 'image-pixels'. We will use this key in the next step.

Step 2: Define the feature columns:

```
>>> ## Step 2: feature columns
>>> image_feature_column = tf.feature_column.numeric_column(
...     key='image-pixels', shape=(28*28))
```

Note that here, we defined the feature columns of size 784 (that is, 28×28), which is the size of the input MNIST images after they are flattened.

Step 3: Create a new Estimator. Here, we specify two hidden layers: 32 units in the first hidden layer and 16 units in the second.

We also specify the number of classes (remember that MNIST consists of 10 different digits, 0-9) using the argument n_classes:

```
>>> ## Step 3: instantiate the estimator
>>> dnn_classifier = tf.estimator.DNNClassifier(
...     feature_columns=[image_feature_column],
...     hidden_units=[32, 16],
...     n_classes=10,
...     model_dir='models/mnist-dnn/')
```

Step 4: Use the Estimator for training, evaluation, and prediction:

```
>>> ## Step 4: train and evaluate
>>> dnn_classifier.train(
...     input_fn=train_input_fn,
...     steps=NUM_EPOCHS * steps_per_epoch)

>>> eval_result = dnn_classifier.evaluate(
...     input_fn=eval_input_fn)

>>> print(eval_result)
{'accuracy': 0.8957, 'average_loss': 0.3876346, 'loss': 0.38815108,
'global_step': 18760}
```

So far, you have learned how to use pre-made Estimators and apply them for preliminary assessment to see, for example, whether an existing model is suitable for a particular problem. Besides using pre-made Estimators, we can also create an Estimator by converting a Keras model to an Estimator, which we will do in the next subsection.

Creating a custom Estimator from an existing Keras model

Converting a Keras model to an Estimator is useful in academia as well as industry for cases where you have developed a model and want to publish it or share the model with other members in your organization. Such a conversion allows us to access the strengths of Estimators, such as distributed training and automatic checkpointing. In addition, it will make it easy for others to use this model, and particularly to avoid confusions in interpreting the input features by specifying the feature columns and the input function.

To learn how we can create our own Estimator from a Keras model, we will work with the previous XOR problem. First, we will regenerate the data and split it into training and validation datasets:

```
>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> ## Create the data
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1]<0] = 0

>>> x_train = x[:100, :]
>>> y_train = y[:100]
>>> x_valid = x[100:, :]
>>> y_valid = y[100:]
```

Let's also build a Keras model that we want to convert to an Estimator later. We will define the model using the `Sequential` class as before. This time, we will also add an input layer defined as `tf.keras.layers.Input` to give a name to the input to this model:

```
>>> model = tf.keras.Sequential([
...     tf.keras.layers.Input(shape=(2,), name='input-features'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(1, activation='sigmoid')
... ])
```

Next, we will go through the four steps that we described in the previous subsection. Steps 1, 2, and 4 will be the same as the ones we used with the pre-made estimators. Note that the key name for the input features that we use in steps 1 and 2 must match with what we defined in the input layer of our model. The code is as follows:

```
>>> ## Step 1: Define the input functions
>>> def train_input_fn(x_train, y_train, batch_size=8):
...     dataset = tf.data.Dataset.from_tensor_slices(
...         ({'input-features':x_train}, y_train.reshape(-1, 1)))
...
...     # shuffle, repeat, and batch the examples.
...     return dataset.shuffle(100).repeat().batch(batch_size)

>>> def eval_input_fn(x_test, y_test=None, batch_size=8):
...     if y_test is None:
...         dataset = tf.data.Dataset.from_tensor_slices(
...             {'input-features':x_test})
...     else:
...         dataset = tf.data.Dataset.from_tensor_slices(
...             ({'input-features':x_test}, y_test.reshape(-1, 1)))
...
...
...     # shuffle, repeat, and batch the examples.
...     return dataset.batch(batch_size)

>>> ## Step 2: Define the feature columns
>>> features = [
...     tf.feature_column.numeric_column(
...         key='input-features:', shape=(2,))
... ]
```

For step 3, we will convert the model to an Estimator using `tf.keras.estimator.model_to_estimator` instead of instantiating one of the pre-made Estimators. Before converting the model, we first need to compile it:

```
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> my_estimator = tf.keras.estimator.model_to_estimator(
...     keras_model=model,
...     model_dir='models/estimator-for-XOR/')
```

Finally, in step 4, we can train our model using the Estimator and evaluate it on the validation dataset:

```
>>> ## Step 4: Use the estimator
>>> num_epochs = 200
>>> batch_size = 2
>>> steps_per_epoch = np.ceil(len(x_train) / batch_size)

>>> my_estimator.train(
...     input_fn=lambda: train_input_fn(x_train, y_train, batch_size),
...     steps=num_epochs * steps_per_epoch)

>>> my_estimator.evaluate(
...     input_fn=lambda: eval_input_fn(x_valid, y_valid, batch_size))
{'binary_accuracy': 0.96, 'loss': 0.081909806, 'global_step': 10000}
```

As you can see, converting a Keras model to an Estimator is very straightforward. Doing this allows us to easily benefit from the various Estimator strengths, such as distributed training and automatically saving the checkpoints during training.

Summary

In this chapter, we covered TensorFlow's most essential and useful features. We started by discussing the migration from TensorFlow v1.x to v2. In particular, we used TensorFlow's dynamic computation graph approach, the so-called eager execution mode, which makes implementing computations more convenient compared to using static graphs. We also covered the semantics of defining TensorFlow `Variable` objects as model parameters, annotating Python functions using the `tf.function` decorator to improve computational efficiency via graph compilation.

After we considered the concept of computing partial derivatives and gradients of arbitrary functions, we covered the Keras API in more detail. It provides us with a user-friendly interface for building more complex deep NN models. Finally, we utilized TensorFlow's `tf.estimator` API to provide a consistent interface that is typically preferred in production environments. We concluded this chapter by converting a Keras model into a custom Estimator.

Now that we have covered the core mechanics of TensorFlow, the next chapter will introduce the concept behind **convolutional neural network** (**CNN**) architectures for deep learning. CNNs are powerful models and have shown great performance in the field of computer vision.

15

Classifying Images with Deep Convolutional Neural Networks

In the previous chapter, we looked in depth at different aspects of the TensorFlow API, you became familiar with tensors and decorating functions, and you learned how to work with TensorFlow Estimators. In this chapter, you will now learn about **convolutional neural networks** (**CNNs**) for image classification. We will start by discussing the basic building blocks of CNNs, using a bottom-up approach. Then, we will take a deeper dive into the CNN architecture and explore how to implement CNNs in TensorFlow. In this chapter, we will cover the following topics:

- Convolution operations in one and two dimensions
- The building blocks of CNN architectures
- Implementing deep CNNs in TensorFlow
- Data augmentation techniques for improving the generalization performance
- Implementing a face image-based CNN classifier for predicting the gender of a person

The building blocks of CNNs

CNNs are a family of models that were originally inspired by how the visual cortex of the human brain works when recognizing objects. The development of CNNs goes back to the 1990s, when Yann LeCun and his colleagues proposed a novel NN architecture for classifying handwritten digits from images (*Handwritten Digit Recognition with a Back-Propagation Network*, *Y. LeCun*, and others, *1989*, published at the *Neural Information Processing Systems (NeurIPS)* conference).

The human visual cortex

The original discovery of how the visual cortex of our brain functions was made by David H. Hubel and Torsten Wiesel in 1959, when they inserted a microelectrode into the primary visual cortex of an anesthetized cat. Then, they observed that brain neurons respond differently after projecting different patterns of light in front of the cat. This eventually led to the discovery of the different layers of the visual cortex. While the primary layer mainly detects edges and straight lines, higher-order layers focus more on extracting complex shapes and patterns.

Due to the outstanding performance of CNNs for image classification tasks, this particular type of feedforward NN gained a lot of attention and led to tremendous improvements in machine learning for computer vision. Several years later, in 2019, Yann LeCun received the Turing award (the most prestigious award in computer science) for his contributions to the field of artificial intelligence (AI), along with two other researchers, Yoshua Bengio and Geoffrey Hinton, whose names you have already encountered in previous chapters.

In the following sections, we will discuss the broader concept of CNNs and why convolutional architectures are often described as "feature extraction layers." Then, we will delve into the theoretical definition of the type of convolution operation that is commonly used in CNNs and walk through examples for computing convolutions in one and two dimensions.

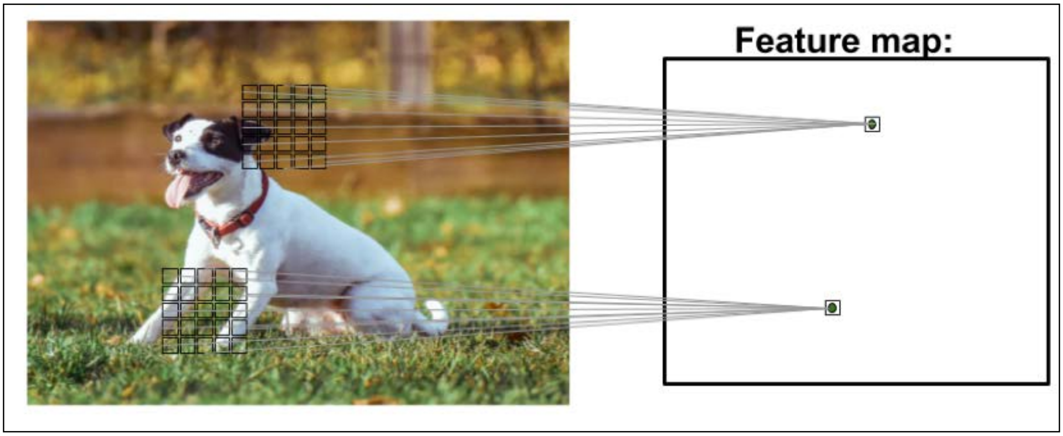
Understanding CNNs and feature hierarchies

Successfully extracting **salient** (**relevant**) **features** is key to the performance of any machine learning algorithm and traditional machine learning models rely on input features that may come from a domain expert or are based on computational feature extraction techniques.

Certain types of NNs, such as CNNs, are able to automatically learn the features from raw data that are most useful for a particular task. For this reason, it's common to consider CNN layers as feature extractors: the early layers (those right after the input layer) extract **low-level features** from raw data, and the later layers (often **fully connected layers** like in a multilayer perceptron (MLP)) use these features to predict a continuous target value or class label.

Certain types of multilayer NNs, and in particular, deep convolutional NNs, construct a so-called **feature hierarchy** by combining the low-level features in a layer-wise fashion to form high-level features. For example, if we're dealing with images, then low-level features, such as edges and blobs, are extracted from the earlier layers, which are combined together to form high-level features. These high-level features can form more complex shapes, such as the general contours of objects like buildings, cats, or dogs.

As you can see in the following image, a CNN computes **feature maps** from an input image, where each element comes from a local patch of pixels in the input image:



(Photo by Alexander Dummer on Unsplash)

This local patch of pixels is referred to as the **local receptive field**. CNNs will usually perform very well on image-related tasks, and that's largely due to two important ideas:

- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image as in the case of perceptrons. You may find it useful to look back and compare how we implemented a fully connected network that connected to the whole image in *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*.)
- **Parameter-sharing**: The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights (parameters) in the network and we will see an improvement in the ability to capture *salient* features. In the context of image data, it makes sense to assume that nearby pixels are typically more relevant to each other than pixels that are far away from each other.

Typically, CNNs are composed of several **convolutional** and subsampling layers that are followed by one or more fully connected layers at the end. The fully connected layers are essentially an MLP, where every input unit, *i*, is connected to every output unit, *j*, with weight w_{ij} (which we covered in more detail in *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*).

Please note that subsampling layers, commonly known as **pooling layers**, do not have any learnable parameters; for instance, there are no weights or bias units in pooling layers. However, both the convolutional and fully connected layers have weights and biases that are optimized during training.

In the following sections, we will study convolutional and pooling layers in more detail and see how they work. To understand how convolution operations work, let's start with a convolution in one dimension, which is sometimes used for working with certain types of sequence data, such as text. After discussing one-dimensional convolutions, we will work through the typical two-dimensional ones that are commonly applied to two-dimensional images.

Performing discrete convolutions

A **discrete convolution** (or simply **convolution**) is a fundamental operation in a CNN. Therefore, it's important to understand how this operation works. In this section, we will cover the mathematical definition and discuss some of the **naive** algorithms to compute convolutions of one-dimensional tensors (vectors) and two-dimensional tensors (matrices).

Please note that the formulas and descriptions in this section are solely for understanding how convolution operations in CNNs work. Indeed, much more efficient implementations of convolutional operations already exist in packages such as TensorFlow, as you will see later in this chapter.

Mathematical notation

In this chapter, we will use subscripts to denote the size of a multidimensional array (tensor); for example, $A_{n_1 \times n_2}$ is a two-dimensional array of size $n_1 \times n_2$. We use brackets, [ ], to denote the indexing of a multidimensional array.

For example, $A[i, j]$ refers to the element at index i, j of matrix $\boldsymbol{A}$. Furthermore, note that we use a special symbol, $*$, to denote the convolution operation between two vectors or matrices, which is not to be confused with the multiplication operator, *, in Python.

Discrete convolutions in one dimension

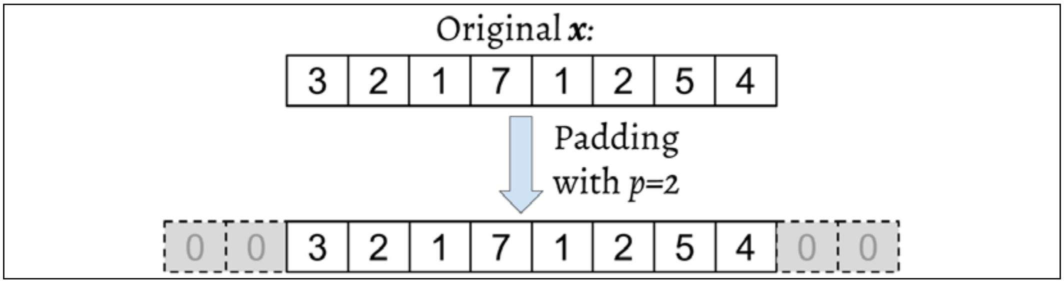
Let's start with some basic definitions and notations that we are going to use. A discrete convolution for two vectors, $\boldsymbol{x}$ and $\boldsymbol{w}$, is denoted by $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, in which vector $\boldsymbol{x}$ is our input (sometimes called **signal**) and $\boldsymbol{w}$ is called the **filter** or **kernel**. A discrete convolution is mathematically defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \;\rightarrow\; y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

As mentioned earlier, the brackets, $[\,]$, are used to denote the indexing for vector elements. The index, i, runs through each element of the output vector, $\boldsymbol{y}$. There are two odd things in the preceding formula that we need to clarify: $-\infty$ to $+\infty$ indices and negative indexing for $\boldsymbol{x}$.

The fact that the sum runs through indices from $-\infty$ to $+\infty$ seems odd, mainly because in machine learning applications, we always deal with finite feature vectors. For example, if $\boldsymbol{x}$ has 10 features with indices 0, 1, 2,…, 8, 9, then indices $-\infty : -1$ and $10 : +\infty$ are out of bounds for $\boldsymbol{x}$. Therefore, to correctly compute the summation shown in the preceding formula, it is assumed that $\boldsymbol{x}$ and $\boldsymbol{w}$ are filled with zeros. This will result in an output vector, $\boldsymbol{y}$, that also has infinite size, with lots of zeros as well. Since this is not useful in practical situations, $\boldsymbol{x}$ is padded only with a finite number of zeros.

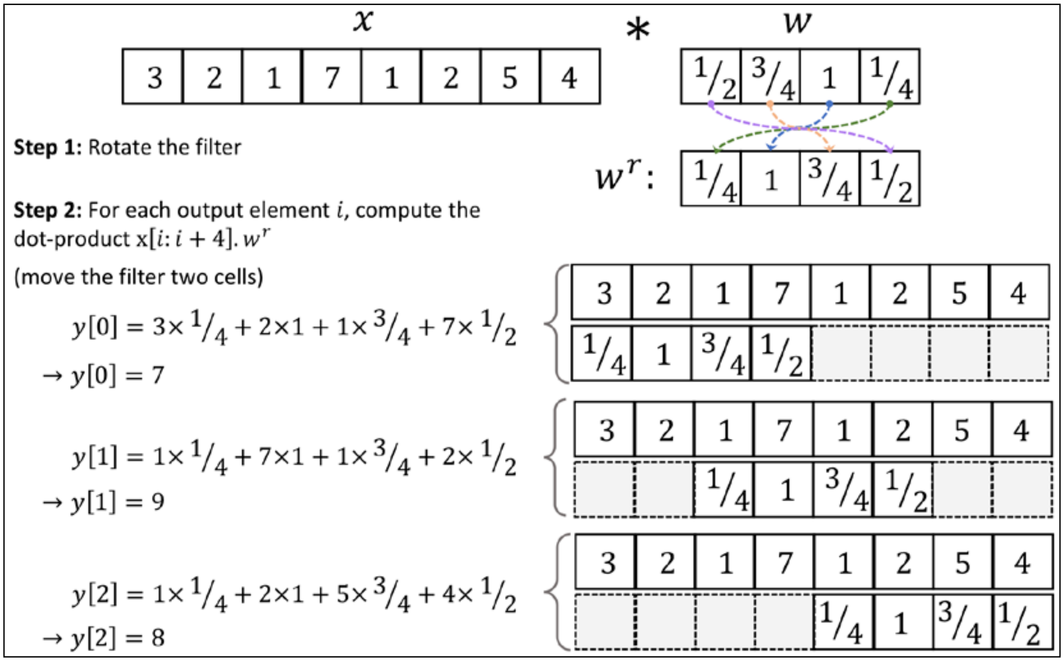
This process is called **zero-padding** or simply **padding**. Here, the number of zeros padded on each side is denoted by p. An example padding of a one-dimensional vector, $\boldsymbol{x}$, is shown in the following figure:



Let's assume that the original input, $\boldsymbol{x}$, and filter, $\boldsymbol{w}$, have n and m elements, respectively, where $m \leq n$. Therefore, the padded vector, $\boldsymbol{x}^p$, has size $n + 2p$. The practical formula for computing a discrete convolution will change to the following:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \;\rightarrow\; y[i] = \sum_{k=0}^{k=m-1} \boldsymbol{x}^p[i+m-k]w[k]$$

Now that we have solved the infinite index issue, the second issue is indexing $\boldsymbol{x}$ with $i + m - k$. The important point to notice here is that $\boldsymbol{x}$ and $\boldsymbol{w}$ are indexed in different directions in this summation. Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors, $\boldsymbol{x}$ or $\boldsymbol{w}$, after they are padded. Then, we can simply compute their dot product. Let's assume we flip (rotate) the filter, $\boldsymbol{w}$, to get the rotated filter, $\boldsymbol{w}^r$. Then, the dot product, $\boldsymbol{x}[i: i+m] \,.\, \boldsymbol{w}^r$, is computed to get one element, $y[i]$, where $\boldsymbol{x}[i: i + m]$ is a patch of $\boldsymbol{x}$ with size m. This operation is repeated like in a sliding window approach to get all the output elements. The following figure provides an example with $\boldsymbol{x}$ = [3 2 1 7 1 2 5 4] and $\boldsymbol{w} = \left[\frac{1}{2}\,\frac{3}{4}\,1\,\frac{1}{4}\right]$ so that the first three output elements are computed:



x
*
w
3 2 1 7 1 2 5 4
1/2 3/4 1 1/4
Step 1: Rotate the filter
w^r: 1/4 1 3/4 1/2
Step 2: For each output element i, compute the dot-product x$[i: i+4]. w^r$
(move the filter two cells)
$y[0] = 3\times 1/4 + 2\times 1 + 1\times 3/4 + 7\times 1/2$
$\rightarrow y[0] = 7$
$y[1] = 1\times 1/4 + 7\times 1 + 1\times 3/4 + 2\times 1/2$
$\rightarrow y[1] = 9$
$y[2] = 1\times 1/4 + 2\times 1 + 5\times 3/4 + 4\times 1/2$
$\rightarrow y[2] = 8$

You can see in the preceding example that the padding size is zero (p = 0). Notice that the rotated filter, $\boldsymbol{w}^r$, is shifted by two cells each time we **shift**. This shift is another hyperparameter of a convolution, the **stride**, s. In this example, the stride is two, s = 2. Note that the stride has to be a positive number smaller than the size of the input vector. We will talk more about padding and strides in the next section.

Cross-correlation

Cross-correlation (or simply correlation) between an input vector and a filter is denoted by $\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w}$ and is very much like a sibling of a convolution, with a small difference: in cross-correlation, the multiplication is performed in the same direction. Therefore, it is not a requirement to rotate the filter matrix, ***w***, in each dimension. Mathematically, cross-correlation is defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w} \rightarrow \; y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]w[k]$$

The same rules for padding and stride may be applied to cross-correlation as well. Note that most deep learning frameworks (including TensorFlow) implement cross-correlation but refer to it as convolution, which is a common convention in the deep learning field.

Padding inputs to control the size of the output feature maps

So far, we've only used zero-padding in convolutions to compute finite-sized output vectors. Technically, padding can be applied with any $p \geq 0$. Depending on the choice of p, boundary cells may be treated differently than the cells located in the middle of ***x***.

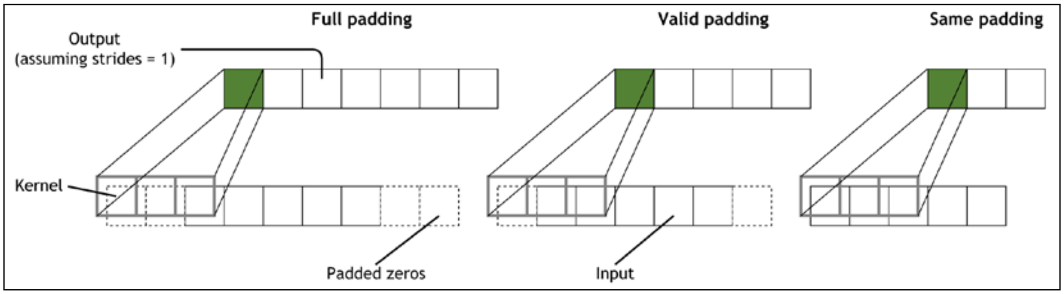
Now, consider an example where n = 5 and m = 3. Then, with p=0, x[0] is only used in computing one output element (for instance, y[0]), while x[1] is used in the computation of two output elements (for instance, y[0] and y[1]). So, you can see that this different treatment of elements of ***x*** can artificially put more emphasis on the middle element, x[2], since it has appeared in most computations. We can avoid this issue if we choose p = 2, in which case, each element of ***x*** will be involved in computing three elements of ***y***.

Furthermore, the size of the output, ***y***, also depends on the choice of the padding strategy we use.

There are three modes of padding that are commonly used in practice: *full*, *same*, and *valid*:

- In full mode, the padding parameter, p, is set to $p = m - 1$. Full padding increases the dimensions of the output; thus, it is rarely used in CNN architectures.
- Same padding is usually used to ensure that the output vector has the same size as the input vector, $\boldsymbol{x}$. In this case, the padding parameter, p, is computed according to the filter size, along with the requirement that the input size and output size are the same.
- Finally, computing a convolution in the valid mode refers to the case where $p = 0$ (no padding).

The following figure illustrates the three different padding modes for a simple 5 × 5 pixel input with a kernel size of 3 × 3 and a stride of 1:



The most commonly used padding mode in CNNs is same padding. One of its advantages over the other padding modes is that same padding preserves the size of the vector—or the height and width of the input images when we are working on image-related tasks in computer vision—which makes designing a network architecture more convenient.

One big disadvantage of valid padding versus full and same padding, for example, is that the volume of the tensors will decrease substantially in NNs with many layers, which can be detrimental to the network performance.

In practice, it is recommended that you preserve the spatial size using same padding for the convolutional layers and decrease the spatial size via pooling layers instead. As for full padding, its size results in an output larger than the input size. Full padding is usually used in signal processing applications where it is important to minimize boundary effects. However, in the deep learning context, boundary effects are usually not an issue, so we rarely see full padding being used in practice.

Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that we shift the filter, $\boldsymbol{w}$, along the input vector. Let's assume that the input vector is of size *n* and the filter is of size *m*. Then, the size of the output resulting from $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, with padding, *p*, and stride, *s*, would be determined as follows:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Here, $\lfloor \cdot \rfloor$ denotes the *floor* operation.

The floor operation

The floor operation returns the largest integer that is equal to or smaller than the input, for example:

$$floor(1.77) = \lfloor 1.77 \rfloor = 1$$

Consider the following two cases:

- Compute the output size for an input vector of size 10 with a convolution kernel of size 5, padding 2, and stride 1:

  $$n = 10, m = 5, \quad p = 2, \quad s = 1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

  (Note that in this case, the output size turns out to be the same as the input; therefore, we can conclude this to be the same-padding mode.)

- How does the output size change for the same input vector when we have a kernel of size 3 and stride 2?

  $$n = 10, m = 3, \quad p = 2, \quad s = 2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

If you are interested in learning more about the size of the convolution output, we recommend the manuscript *A guide to convolution arithmetic for deep learning*, by *Vincent Dumoulin* and *Francesco Visin*, which is freely available at `https://arxiv.org/abs/1603.07285`.

Finally, in order to learn how to compute convolutions in one dimension, a naive implementation is shown in the following code block, and the results are compared with the `numpy.convolve` function. The code is as follows:

```
>>> import numpy as np
>>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     if p > 0:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([zero_pad,
...                                    x_padded,
...                                    zero_pad])
...     res = []
...     for i in range(0, int(len(x)/s),s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] *
...                           w_rot))
...     return np.array(res)

>>> ## Testing:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]

>>> print('Conv1d Implementation:',
...       conv1d(x, w, p=2, s=1))
Conv1d Implementation: [ 5. 14. 16. 26. 24. 34. 19. 22.]

>>> print('NumPy Results:',
...       np.convolve(x, w, mode='same'))
NumPy Results: [ 5 14 16 26 24 34 19 22]
```

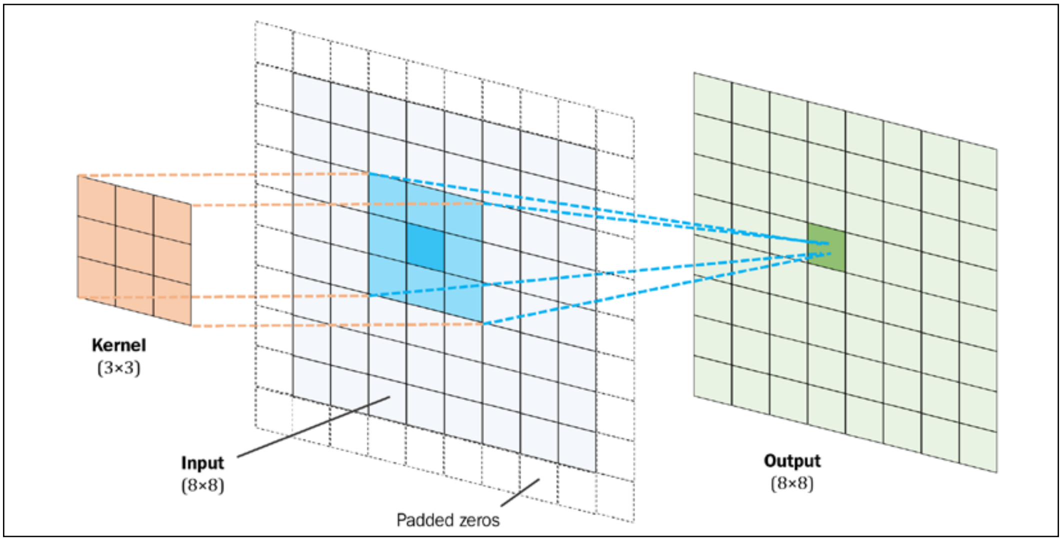
So far, we have mostly focused on convolutions for vectors (1D convolutions). We started with the 1D case to make the concepts easier to understand. In the next section, we will cover 2D convolutions in more detail, which are the building blocks of CNNs for image-related tasks.

Performing a discrete convolution in 2D

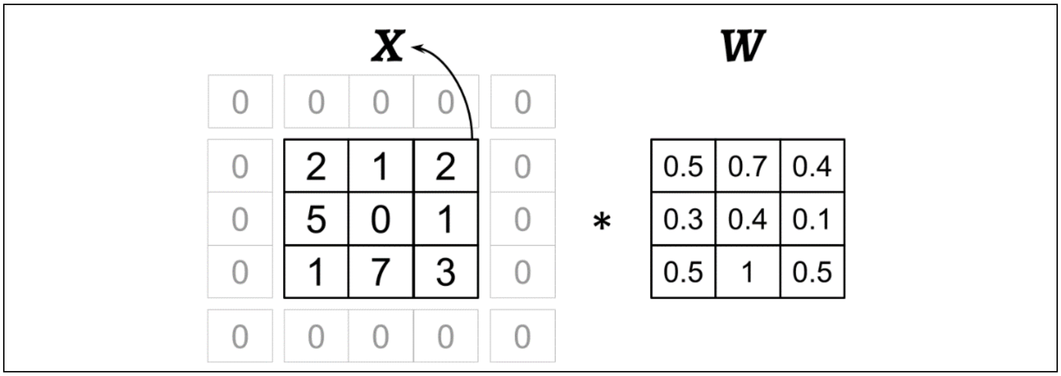
The concepts you learned in the previous sections are easily extendible to 2D. When we deal with 2D inputs, such as a matrix, $\boldsymbol{X}_{n_1 \times n_2}$, and the filter matrix, $\boldsymbol{W}_{m_1 \times m_2}$, where $m_1 \le n_1$ and $m_2 \le n_2$, then the matrix $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$ is the result of a 2D convolution between *X* and *W*. This is defined mathematically as follows:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \;\rightarrow\; Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2]\, W[k_1, k_2]$$

Notice that if you omit one of the dimensions, the remaining formula is exactly the same as the one we used previously to compute the convolution in 1D. In fact, all the previously mentioned techniques, such as zero-padding, rotating the filter matrix, and the use of strides, are also applicable to 2D convolutions, provided that they are extended to both dimensions independently. The following figure demonstrates 2D convolution of an input matrix of size 8×8, using a kernel of size 3×3. The input matrix is padded with zeros with $p = 1$. As a result, the output of the 2D convolution will have a size of 8×8:



The following example illustrates the computation of a 2D convolution between an input matrix, $\boldsymbol{X}_{3\times3}$, and a kernel matrix, $\boldsymbol{W}_{3\times3}$, using padding $p = (1, 1)$ and stride $s = (2, 2)$. According to the specified padding, one layer of zeros is added on each side of the input matrix, which results in the padded matrix $\boldsymbol{X}^{\text{padded}}_{5\times5}$, as follows:

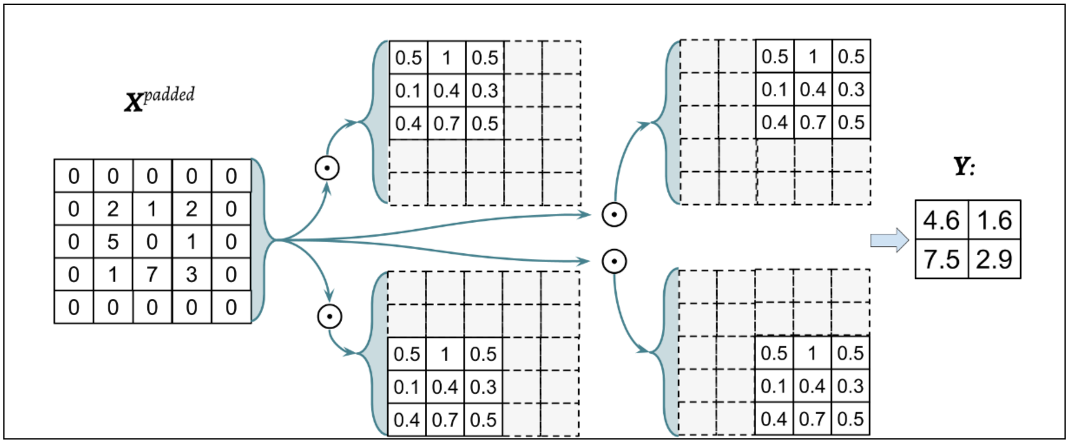


$$\boldsymbol{X}: \begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 1 & 7 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \quad * \quad \boldsymbol{W}: \begin{array}{|c|c|c|} \hline 0.5 & 0.7 & 0.4 \\ \hline 0.3 & 0.4 & 0.1 \\ \hline 0.5 & 1 & 0.5 \\ \hline \end{array}$$

With the preceding filter, the rotated filter will be:

$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Note that this rotation is not the same as the transpose matrix. To get the rotated filter in NumPy, we can write `W_rot=W[::-1,::-1]`. Next, we can shift the rotated filter matrix along the padded input matrix, $\boldsymbol{X}^{\text{padded}}$, like a sliding window and compute the sum of the element-wise product, which is denoted by the $\odot$ operator in the following figure:



The result will be the 2 × 2 matrix, $\boldsymbol{Y}$.

Let's also implement the 2D convolution according to the *naive* algorithm described. The `scipy.signal` package provides a way to compute 2D convolution via the `scipy.signal.convolve2d` function:

```
>>> import numpy as np
>>> import scipy.signal

>>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[::-1,::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0]
...     n2 = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
...     X_padded[p[0]:p[0]+X_orig.shape[0],
...              p[1]:p[1]+X_orig.shape[1]] = X_orig
...
```

```
...     res = []
...     for i in range(0, int((X_padded.shape[0] - \
...                               W_rot.shape[0])/s[0])+1, s[0]):
...         res.append([])
...         for j in range(0, int((X_padded.shape[1] - \
...                                   W_rot.shape[1])/s[1])+1, s[1]):
...             X_sub = X_padded[i:i+W_rot.shape[0],
...                                  j:j+W_rot.shape[1]]
...             res[-1].append(np.sum(X_sub * W_rot))
...     return(np.array(res))
>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]

>>> print('Conv2d Implementation:\n',
...       conv2d(X, W, p=(1, 1), s=(1, 1)))
Conv2d Implementation:
[[ 11.  25.  32.  13.]
 [ 19.  25.  24.  13.]
 [ 13.  28.  25.  17.]
 [ 11.  17.  14.   9.]]


>>> print('SciPy Results:\n',
...       scipy.signal.convolve2d(X, W, mode='same'))
SciPy Results:
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

Efficient algorithms for computing convolution

We provided a naive implementation to compute a 2D convolution for the purpose of understanding the concepts. However, this implementation is very inefficient in terms of memory requirements and computational complexity. Therefore, it should not be used in real-world NN applications.

One aspect is that the filter matrix is actually not rotated in most tools like TensorFlow. Moreover, in recent years, much more efficient algorithms have been developed that use the Fourier transform to compute convolutions. It is also important to note that in the context of NNs, the size of a convolution kernel is usually much smaller than the size of the input image.

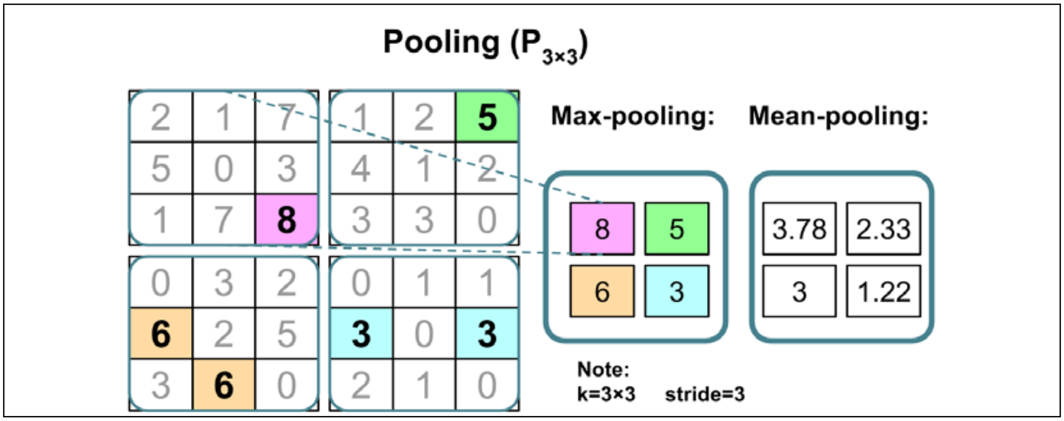
For example, modern CNNs usually use kernel sizes such as 1 × 1, 3 × 3, or 5 × 5, for which efficient algorithms have been designed that can carry out the convolutional operations much more efficiently, such as **Winograd's minimal filtering** algorithm. These algorithms are beyond the scope of this book, but if you are interested in learning more, you can read the manuscript *Fast Algorithms for Convolutional Neural Networks*, by *Andrew Lavin* and *Scott Gray*, 2015, which is freely available at `https://arxiv.org/abs/1509.09308`.

In the next section, we will discuss subsampling or pooling, which is another important operation often used in CNNs.

Subsampling layers

Subsampling is typically applied in two forms of pooling operations in CNNs: **max-pooling** and **mean-pooling** (also known as **average-pooling**). The pooling layer is usually denoted by $P_{n_1 \times n_2}$. Here, the subscript determines the size of the neighborhood (the number of adjacent pixels in each dimension) where the max or mean operation is performed. We refer to such a neighborhood as the **pooling size**.

The operation is described in the following figure. Here, max-pooling takes the maximum value from a neighborhood of pixels, and mean-pooling computes their average:



The advantage of pooling is twofold:

- Pooling (max-pooling) introduces a local invariance. This means that small changes in a local neighborhood do not change the result of max-pooling.

Therefore, it helps with generating features that are more robust to noise in the input data. Refer to the following example, which shows that the max-pooling of two different input matrices, $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, results in the same output:

$$\left.\begin{matrix} \boldsymbol{X}_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix} \\ \\ \boldsymbol{X}_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix} \end{matrix}\right\} \xrightarrow{\text{max pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

- Pooling decreases the size of features, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well.

Overlapping versus non-overlapping pooling

Traditionally, pooling is assumed to be non-overlapping. Pooling is typically performed on non-overlapping neighborhoods, which can be done by setting the stride parameter equal to the pooling size. For example, a non-overlapping pooling layer, $P_{n_1 \times n_2}$, requires a stride parameter $s = (n_1, n_2)$. On the other hand, overlapping pooling occurs if the stride is smaller than the pooling size. An example where overlapping pooling is used in a convolutional network is described in *ImageNet Classification with Deep Convolutional Neural Networks*, by *A. Krizhevsky*, *I. Sutskever*, and *G. Hinton*, 2012, which is freely available as a manuscript at `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks`.

While pooling is still an essential part of many CNN architectures, several CNN architectures have also been developed without using pooling layers. Instead of using pooling layers to reduce the feature size, researchers use convolutional layers with a stride of 2.

In a sense, you can think of a convolutional layer with stride 2 as a pooling layer with learnable weights. If you are interested in an empirical comparison of different CNN architectures developed with and without pooling layers, we recommend reading the research article *Striving for Simplicity: The All Convolutional Net*, by *Jost Tobias Springenberg*, *Alexey Dosovitskiy*, *Thomas Brox*, and *Martin Riedmiller*. This article is freely available at `https://arxiv.org/abs/1412.6806`.

Putting everything together – implementing a CNN

So far, you have learned about the basic building blocks of CNNs. The concepts illustrated in this chapter are not really more difficult than traditional multilayer NNs. We can say that the most important operation in a traditional NN is matrix multiplication. For instance, we use matrix multiplications to compute the pre-activations (or net inputs), as in $\boldsymbol{z} = \boldsymbol{Wx} + b$. Here, $\boldsymbol{x}$ is a column vector ($\mathbb{R}^{n \times 1}$ matrix) representing pixels, and $\boldsymbol{W}$ is the weight matrix connecting the pixel inputs to each hidden unit.

In a CNN, this operation is replaced by a convolution operation, as in $\boldsymbol{Z} = \boldsymbol{W} * \boldsymbol{X} + \boldsymbol{b}$, where $\boldsymbol{X}$ is a matrix representing the pixels in a $height \times width$ arrangement. In both cases, the pre-activations are passed to an activation function to obtain the activation of a hidden unit, $\boldsymbol{A} = \phi(\boldsymbol{Z})$, where ϕ is the activation function. Furthermore, you will recall that subsampling is another building block of a CNN, which may appear in the form of pooling, as was described in the previous section.

Working with multiple input or color channels

An input to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 \times N_2$ (for example, the image height and width in pixels). These $N_1 \times N_2$ matrices are called *channels*. Conventional implementations of convolutional layers expect a rank-3 tensor representation as an input, for example a three-dimensional array, $\boldsymbol{X}_{N_1 \times N_2 \times C_{in}}$, where C_{in} is the number of input channels. For example, let's consider images as input to the first layer of a CNN. If the image is colored and uses the RGB color mode, then $C_{in} = 3$ (for the red, green, and blue color channels in RGB). However, if the image is in grayscale, then we have $C_{in} = 1$, because there is only one channel with the grayscale pixel intensity values.

Reading an image file

When we work with images, we can read images into NumPy arrays using the `uint8` (unsigned 8-bit integer) data type to reduce memory usage compared to 16-bit, 32-bit, or 64-bit integer types, for example.

Unsigned 8-bit integers take values in the range [0, 255], which are sufficient to store the pixel information in RGB images, which also take values in the same range.

In *Chapter 13, Parallelizing Neural Network Training with TensorFlow*, you saw that TensorFlow provides a module for loading/storing and manipulating images via `tf.io` and `tf.image` submodules. Let's recap how to read an image (this example RGB image is located in the code bundle folder that is provided with this chapter at `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/code/ch15`):

```
>>> import tensorflow as tf
>>> img_raw = tf.io.read_file('example-image.png')
>>> img = tf.image.decode_image(img_raw)
>>> print('Image shape:', img.shape)
Image shape: (252, 221, 3)
```

When you build models and data loaders in TensorFlow, it is recommended that you use `tf.image` as well to read in the input images.

Now, let's also look at an example of how we can read in an image into our Python session using the `imageio` package. We can install `imageio` either via `conda` or `pip` from the command-line terminal:

```
> conda install imageio
```

or

```
> pip install imageio
```

Once `imageio` is installed, we can use the `imread` function to read in the same image we used previously by using the `imageio` package:

```
>>> import imageio
>>> img = imageio.imread('example-image.png')
>>> print('Image shape:', img.shape)
Image shape: (252, 221, 3)
>>> print('Number of channels:', img.shape[2])
Number of channels: 3
>>> print('Image data type:', img.dtype)
Image data type: uint8
>>> print(img[100:102, 100:102, :])
[[[179 134 110]
  [182 136 112]]
 [[180 135 11]
  [182 137 113]]]
```

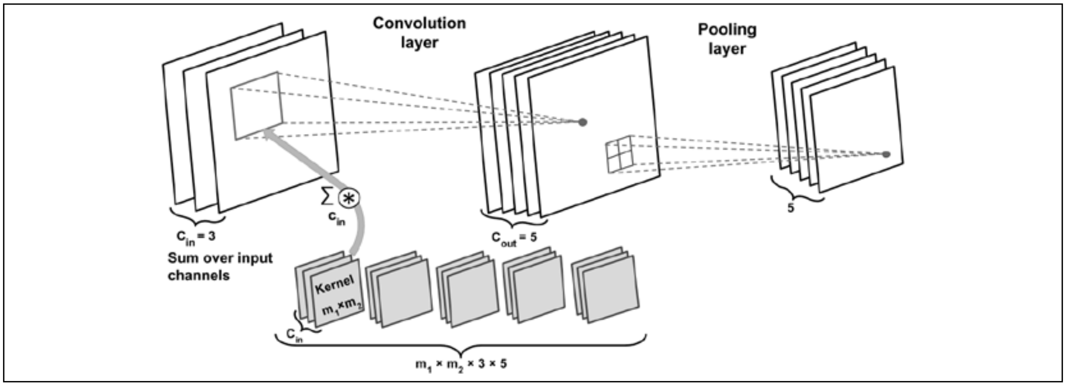
Now that you are familiar with the structure of input data, the next question is, how can we incorporate multiple input channels in the convolution operation that we discussed in the previous sections? The answer is very simple: we perform the convolution operation for each channel separately and then add the results together using the matrix summation. The convolution associated with each channel (*c*) has its own kernel matrix as *W*[:, :, *c*]. The total pre-activation result is computed in the following formula:

$$\begin{array}{l}\text{Given an example } \boldsymbol{X}_{n_1\times n_2\times C_{in}},\\ \text{a kernel matrix } \boldsymbol{W}_{m_1\times m_2\times C_{in}},\\ \text{and bias value } b\end{array} \implies \begin{cases} \boldsymbol{Z}^{Conv} = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c] * \boldsymbol{X}[:,:,c] \\ \text{Pre-activation:} \quad \boldsymbol{Z} = \boldsymbol{Z}^{Conv} + b_C \\ \text{Feature map:} \quad \boldsymbol{A} = \phi(\boldsymbol{Z}) \end{cases}$$

The final result, *A*, is a feature map. Usually, a convolutional layer of a CNN has more than one feature map. If we use multiple feature maps, the kernel tensor becomes four-dimensional: $width \times height \times C_{in} \times C_{out}$. Here, $width \times height$ is the kernel size, C_{in} is the number of input channels, and C_{out} is the number of output feature maps. So, now let's include the number of output feature maps in the preceding formula and update it, as follows:

$$\begin{array}{l}\text{Given an example } \boldsymbol{X}_{n_1\times n_2\times C_{in}},\\ \text{a kernel matrix } \boldsymbol{W}_{m_1\times m_2\times C_{in}\times C_{out}},\\ \text{and bias vector } \boldsymbol{b}_{C_{out}}\end{array} \implies \begin{cases} \boldsymbol{Z}^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c,k] * \boldsymbol{X}[:,:,c] \\ \boldsymbol{Z}[:,:,k] = \boldsymbol{Z}^{Conv}[:,:,k] + b[k] \\ \boldsymbol{A}[:,:,k] = \phi(\boldsymbol{Z}[:,:,k]) \end{cases}$$

To conclude our discussion of computing convolutions in the context of NNs, let's look at the example in the following figure, which shows a convolutional layer, followed by a pooling layer. In this example, there are three input channels. The kernel tensor is four-dimensional. Each kernel matrix is denoted as $m_1 \times m_2$, and there are three of them, one for each input channel. Furthermore, there are five such kernels, accounting for five output feature maps. Finally, there is a pooling layer for subsampling the feature maps:



How many trainable parameters exist in the preceding example?

To illustrate the advantages of convolution, **parameter-sharing**, and **sparse connectivity**, let's work through an example. The convolutional layer in the network shown in the preceding figure is a four-dimensional tensor. So, there are $m_1 \times m_2 \times 3 \times 5$ parameters associated with the kernel. Furthermore, there is a bias vector for each output feature map of the convolutional layer. Thus, the size of the bias vector is 5. Pooling layers do not have any (trainable) parameters; therefore, we can write the following:

$$m_1 \times m_2 \times 3 \times 5 + 5$$

If the input tensor is of size $n_1 \times n_2 \times 3$, assuming that the convolution is performed with the same-padding mode, then the output feature maps would be of size $n_1 \times n_2 \times 5$.

Note that if we use a fully connected layer instead of a convolutional layer, this number will be much larger. In the case of a fully connected layer, the number of parameters for the weight matrix to reach the same number of output units would have been as follows:

$$(n_1 \times n_2 \times 3)\ \times\ (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5$$

In addition, the size of the bias vector is $n_1 \times n_2 \times 5$ (one bias element for each output unit). Given that $m_1 < n_1$ and $m_2 < n_2$, we can see that the difference in the number of trainable parameters is significant.

Lastly, as was already mentioned, typically, the convolution operations are carried out by treating an input image with multiple color channels as a stack of matrices; that is, we perform the convolution on each matrix separately and then add the results, as was illustrated in the previous figure. However, convolutions can also be extended to 3D volumes if you are working with 3D datasets, for example, as shown in the paper *VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition* (2015), by *Daniel Maturana* and *Sebastian Scherer*, which can be accessed at `https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf`.

In the next section, we will talk about how to regularize an NN.

Regularizing an NN with dropout

Choosing the size of a network, whether we are dealing with a traditional (fully connected) NN or a CNN, has always been a challenging problem. For instance, the size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance.

You will recall from *Chapter 14*, *Going Deeper – The Mechanics of TensorFlow*, that a simple network without any hidden layer could only capture a linear decision boundary, which is not sufficient for dealing with an exclusive or (XOR) or similar problem. The *capacity* of a network refers to the level of complexity of the function that it can learn to approximate. Small networks, or networks with a relatively small number of parameters, have a low capacity and are therefore likely to *underfit*, resulting in poor performance, since they cannot learn the underlying structure of complex datasets. However, very large networks may result in *overfitting*, where the network will memorize the training data and do extremely well on the training dataset while achieving a poor performance on the held-out test dataset. When we deal with real-world machine learning problems, we do not know how large the network should be *a priori*.

One way to address this problem is to build a network with a relatively large capacity (in practice, we want to choose a capacity that is slightly larger than necessary) to do well on the training dataset. Then, to prevent overfitting, we can apply one or multiple regularization schemes to achieve a good generalization performance on new data, such as the held-out test dataset.

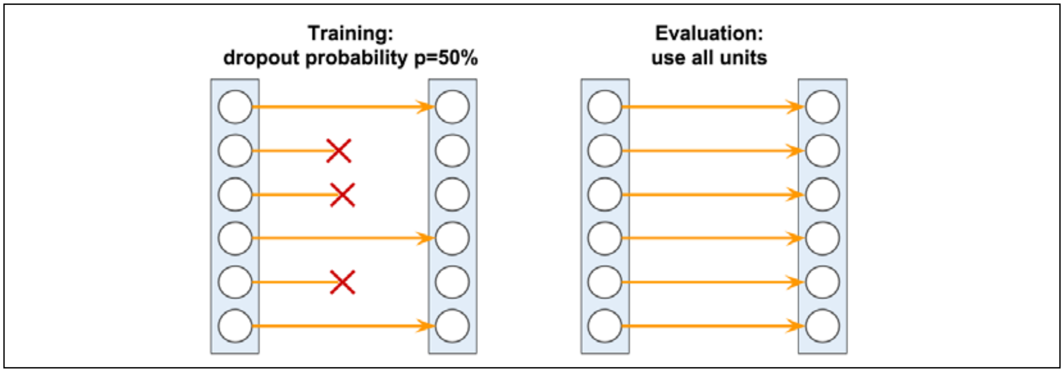
In *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, we covered L1 and L2 regularization. In the section *Tackling overfitting via regularization*, you saw that both techniques, L1 and L2 regularization, can prevent or reduce the effect of overfitting by adding a penalty to the loss that results in shrinking the weight parameters during training. While both L1 and L2 regularization can be used for NNs as well, with L2 being the more common choice of the two, there are other methods for regularizing NNs, such as dropout, which we discuss in this section. But before we move on to discussing dropout, to use L2 regularization within a convolutional or fully connected (dense) network, you can simply add the L2 penalty to the loss function by setting the `kernel_regularizer` of a particular layer when using the Keras API, as follows (it will then automatically modify the loss function accordingly):

```
>>> from tensorflow import keras

>>> conv_layer = keras.layers.Conv2D(
...     filters=16,
...     kernel_size=(3,3),
...     kernel_regularizer=keras.regularizers.l2(0.001))

>>> fc_layer = keras.layers.Dense(
...     units=16,
...     kernel_regularizer=keras.regularizers.l2(0.001))
```

In recent years, **dropout** has emerged as a popular technique for regularizing (deep) NNs to avoid overfitting, thus improving the generalization performance (*Dropout: a simple way to prevent neural networks from overfitting*, by *N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov*, *Journal of Machine Learning Research 15.1*, pages 1929-1958, 2014, `http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`). Dropout is usually applied to the hidden units of higher layers and works as follows: during the training phase of an NN, a fraction of the hidden units is randomly dropped at every iteration with probability p_{drop} (or keep probability $p_{\text{keep}} = 1 - p_{\text{drop}}$). This dropout probability is determined by the user and the common choice is $p = 0.5$, as discussed in the previously mentioned article by Nitish Srivastava and others, 2014. When dropping a certain fraction of input neurons, the weights associated with the remaining neurons are rescaled to account for the missing (dropped) neurons.

The effect of this random dropout is that the network is forced to learn a redundant representation of the data. Therefore, the network cannot rely on an activation of any set of hidden units, since they may be turned off at any time during training, and is forced to learn more general and robust patterns from the data.

This random dropout can effectively prevent overfitting. The following figure shows an example of applying dropout with probability $p = 0.5$ during the training phase, whereby half of the neurons will become inactive randomly (dropped units are selected randomly in each forward pass of training). However, during prediction, all neurons will contribute to computing the pre-activations of the next layer.



As shown here, one important point to remember is that units may drop randomly during training only, whereas for the evaluation (inference) phase, all the hidden units must be active (for instance, $p_{\text{drop}} = 0$ or $p_{\text{keep}} = 1$). To ensure that the overall activations are on the same scale during training and prediction, the activations of the active neurons have to be scaled appropriately (for example, by halving the activation if the dropout probability was set to $p = 0.5$).

However, since it is inconvenient to always scale activations when making predictions, TensorFlow and other tools scale the activations during training (for example, by doubling the activations if the dropout probability was set to $p = 0.5$). This approach is commonly referred to as inverse dropout.

While the relationship is not immediately obvious, dropout can be interpreted as the consensus (averaging) of an ensemble of models. As discussed in *Chapter 7, Combining Different Models for Ensemble Learning*, in ensemble learning, we train several models independently. During prediction, we then use the consensus of all the trained models. We already know that model ensembles are known to perform better than single models. In deep learning, however, both training several models and collecting and averaging the output of multiple models is computationally expensive. Here, dropout offers a workaround, with an efficient way to train many models at once and compute their average predictions at test or prediction time.

As mentioned previously, the relationship between model ensembles and dropout is not immediately obvious. However, consider that in dropout, we have a different model for each mini-batch (due to setting the weights to zero randomly during each forward pass).

Then, via iterating over the mini-batches, we essentially sample over $M = 2^h$ models, where h is the number of hidden units.

The restriction and aspect that distinguishes dropout from regular ensembling, however, is that we share the weights over these "different models", which can be seen as a form of regularization. Then, during "inference" (for instance, predicting the labels in the test dataset), we can average over all these different models that we sampled over during training. This is very expensive, though.

Then, averaging the models, that is, computing the geometric mean of the class-membership probability that is returned by a model, i, can be computed as follows:

$$p_{Ensemble} = \left[\prod_{j=1}^{M} p^{\{i\}} \right]^{\frac{1}{M}}$$

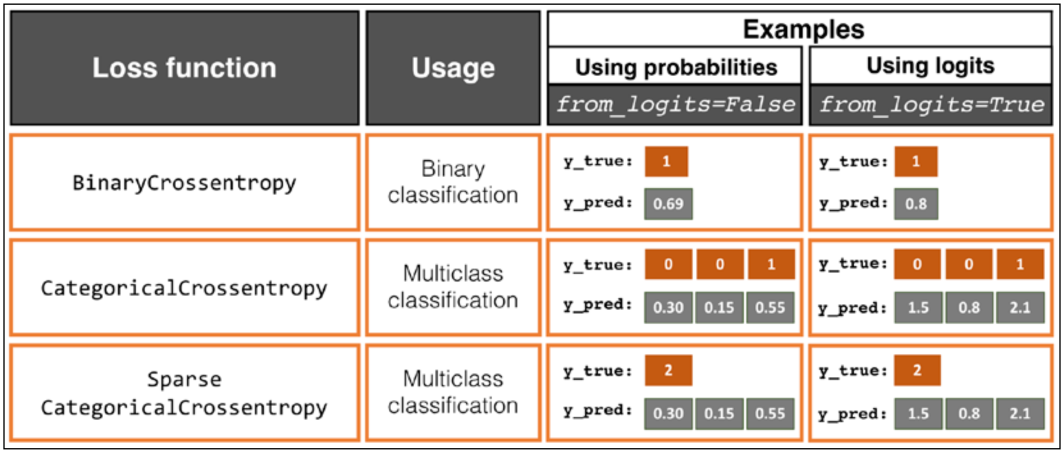
Now, the trick behind dropout is that this geometric mean of the model ensembles (here, *M* models) can be approximated by scaling the predictions of the last (or final) model sampled during training by a factor of 1/(1 – *p*), which is much cheaper than computing the geometric mean explicitly using the previous equation. (In fact, the approximation is exactly equivalent to the true geometric mean if we consider linear models.)

Loss functions for classification

In *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, we saw different activation functions, such as ReLU, sigmoid, and tanh. Some of these activation functions, like ReLU, are mainly used in the intermediate (hidden) layers of an NN to add non-linearities to our model. But others, like sigmoid (for binary) and softmax (for multiclass), are added at the last (output) layer, which results in class-membership probabilities as the output of the model. If the sigmoid or softmax activations are not included at the output layer, then the model will compute the logits instead of the class-membership probabilities.

Focusing on classification problems here, depending on the type of problem (binary versus multiclass) and the type of output (logits versus probabilities), we should choose the appropriate loss function to train our model. **Binary cross-entropy** is the loss function for a binary classification (with a single output unit), and **categorical cross-entropy** is the loss function for multiclass classification. In the Keras API, two options for categorical cross-entropy loss are provided, depending on whether the ground truth labels are in a one-hot encoded format (for example, [0, 0, 1, 0]), or provided as integer labels (for example, *y*=2), which is also known as "sparse" representation in the context of Keras.

The following table describes three loss functions available in Keras for dealing with all three cases: binary classification, multiclass with one-hot encoded ground truth labels, and multiclass with integer (sparse) labels. Each one of these three loss functions also has the option to receive the predictions in the form of logits or class-membership probabilities:



Loss function	Usage	Examples	
		Using probabilities	Using logits
		from_logits=False	*from_logits=True*
BinaryCrossentropy	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

Please note that computing the cross-entropy loss by providing the logits, and not the class-membership probabilities, is usually preferred due to numerical stability reasons. If we provide logits as inputs to the loss function and set `from_logits=True`, the respective TensorFlow function uses a more efficient implementation to compute the loss and derivative of the loss with respect to the weights. This is possible since certain mathematical terms cancel and thus don't have to be computed explicitly when providing logits as inputs.

The following code will show you how to use these three loss functions with two different formats, where either the logits or class-membership probabilities are given as inputs to the loss functions:

```
>>> import tensorflow_datasets as tfds

>>> ####### Binary Crossentropy
>>> bce_probas = tf.keras.losses.BinaryCrossentropy(from_logits=False)
>>> bce_logits = tf.keras.losses.BinaryCrossentropy(from_logits=True)

>>> logits = tf.constant([0.8])
>>> probas = tf.keras.activations.sigmoid(logits)

>>> tf.print(
...     'BCE (w Probas): {:.4f}'.format(
...     bce_probas(y_true=[1], y_pred=probas)),
...     '(w Logits): {:.4f}'.format(
...     bce_logits(y_true=[1], y_pred=logits)))
BCE (w Probas): 0.3711 (w Logits): 0.3711

>>> ####### Categorical Crossentropy
>>> cce_probas = tf.keras.losses.CategoricalCrossentropy(
...     from_logits=False)
>>> cce_logits = tf.keras.losses.CategoricalCrossentropy(
...     from_logits=True)

>>> logits = tf.constant([[1.5, 0.8, 2.1]])
>>> probas = tf.keras.activations.softmax(logits)

>>> tf.print(
...     'CCE (w Probas): {:.4f}'.format(
...     cce_probas(y_true=[0, 0, 1], y_pred=probas)),
...     '(w Logits): {:.4f}'.format(
...     cce_logits(y_true=[0, 0, 1], y_pred=logits)))
CCE (w Probas): 0.5996 (w Logits): 0.5996

>>> ####### Sparse Categorical Crossentropy
>>> sp_cce_probas = tf.keras.losses.SparseCategoricalCrossentropy(
...     from_logits=False)
>>> sp_cce_logits = tf.keras.losses.SparseCategoricalCrossentropy(
...     from_logits=True)

>>> tf.print(
...     'Sparse CCE (w Probas): {:.4f}'.format(
...     sp_cce_probas(y_true=[2], y_pred=probas)),
...     '(w Logits): {:.4f}'.format(
...     sp_cce_logits(y_true=[2], y_pred=logits)))
Sparse CCE (w Probas): 0.5996 (w Logits): 0.5996
```

Note that sometimes, you may come across an implementation where a categorical cross-entropy loss is used for binary classification. Typically, when we have a binary classification task, the model returns a single output value for each example. We interpret this single model output as the probability of the positive class (for example, class 1), P[class = 1]. In a binary classification problem, it is implied that P[class = 0] = 1 – P[class = 1]; hence, we do not need a second output unit in order to obtain the probability of the negative class. However, sometimes practitioners choose to return two outputs for each training example and interpret them as probabilities of each class: P[class = 0] versus P[class = 1]. Then, in such a case, using a softmax function (instead of the logistic sigmoid) to normalize the outputs (so that they sum to 1) is recommended, and categorical cross-entropy is the appropriate loss function.

Implementing a deep CNN using TensorFlow

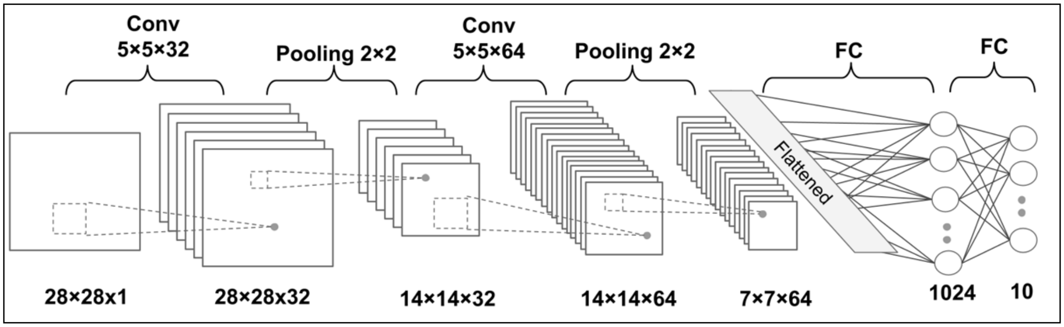
In *Chapter 14*, *Going Deeper – The Mechanics of TensorFlow*, you may recall that we used TensorFlow Estimators for handwritten digit recognition problems, using different API levels of TensorFlow. You may also recall that we achieved about 89 percent accuracy using the `DNNClassifier` Estimator with two hidden layers.

Now, let's implement a CNN and see whether it can achieve a better predictive performance compared to the MLP (`DNNClassifier`) for classifying handwritten digits. Note that the fully connected layers that we saw in *Chapter 14*, *Going Deeper – The Mechanics of TensorFlow*, were able to perform well on this problem. However, in some applications, such as reading bank account numbers from handwritten digits, even tiny mistakes can be very costly. Therefore, it is crucial to reduce this error as much as possible.

The multilayer CNN architecture

The architecture of the network that we are going to implement is shown in the following figure. The inputs are 28 × 28 grayscale images. Considering the number of channels (which is 1 for grayscale images) and a batch of input images, the input tensor's dimensions will be $batchsize \times 28 \times 28 \times 1$.

The input data goes through two convolutional layers that have a kernel size of 5 × 5. The first convolution has 32 output feature maps, and the second one has 64 output feature maps. Each convolution layer is followed by a subsampling layer in the form of a max-pooling operation, $P_{2\times 2}$. Then a fully connected layer passes the output to a second fully connected layer, which acts as the final *softmax* output layer. The architecture of the network that we are going to implement is shown in the following figure:



The dimensions of the tensors in each layer are as follows:

- Input: $[batchsize \times 28 \times 28 \times 1]$
- Conv_1: $[batchsize \times 28 \times 28 \times 32]$
- Pooling_1: $[batchsize \times 14 \times 14 \times 32]$
- Conv_2: $[batchsize \times 14 \times 14 \times 64]$
- Pooling_2: $[batchsize \times 7 \times 7 \times 64]$
- FC_1: $[batchsize \times 1024]$
- FC_2 and softmax layer: $[batchsize \times 10]$

For the convolutional kernels, we are using `strides=1` such that the input dimensions are preserved in the resulting feature maps. For the pooling layers, we are using `strides=2` to subsample the image and shrink the size of the output feature maps. We will implement this network using the TensorFlow Keras API.

Loading and preprocessing the data

You will recall that in *Chapter 13, Parallelizing Neural Network Training with TensorFlow*, you learned two ways of loading available datasets from the `tensorflow_datasets` module. One approach is based on a three-step process, and a simpler method uses a function called `load`, which wraps those three steps. Here, we will use the first method. The three steps for loading the MNIST dataset are as follows:

```
>>> import tensorflow_datasets as tfds
>>> ## Loading the data
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> datasets = mnist_bldr.as_dataset(shuffle_files=False)
>>> mnist_train_orig = datasets['train']
>>> mnist_test_orig = datasets['test']
```

The MNIST dataset comes with a pre-specified training and test dataset partitioning scheme, but we also want to create a validation split from the train partition. Notice that in the third step, we used an optional argument, `shuffle_files=False`, in the `.as_dataset()` method. This prevented initial shuffling, which is necessary for us since we want to split the training dataset into two parts: a smaller training dataset and a validation dataset. (Note: if the initial shuffling was not turned off, it would incur reshuffling of the dataset every time we fetched a mini-batch of data.

An example of this behavior is shown in the online contents of this chapter, where you can see that the number of labels in the validation datasets changes due to reshuffling of the train/validation splits. This can cause *false* performance estimation of the model, since the train/validation datasets are indeed mixed.) We can split the train/validation datasets as follows:

```
>>> BUFFER_SIZE = 10000
>>> BATCH_SIZE = 64
>>> NUM_EPOCHS = 20

>>> mnist_train = mnist_train_orig.map(
...     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
...                   tf.cast(item['label'], tf.int32)))

>>> mnist_test = mnist_test_orig.map(
...     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
...                   tf.cast(item['label'], tf.int32)))

>>> tf.random.set_seed(1)
>>> mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
...                   reshuffle_each_iteration=False)

>>> mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
>>> mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

Now, after preparing the dataset, we are ready to implement the CNN we just described.

Implementing a CNN using the TensorFlow Keras API

For implementing a CNN in TensorFlow, we use the Keras `Sequential` class to stack different layers, such as convolution, pooling, and dropout, as well as the fully connected (dense) layers. The Keras layers API provides classes for each one: `tf.keras.layers.Conv2D` for a two-dimensional convolution layer; `tf.keras.layers.MaxPool2D` and `tf.keras.layers.AvgPool2D` for subsampling (max-pooling and average-pooling); and `tf.keras.layers.Dropout` for regularization using dropout. We will go over each of these classes in more detail.

Configuring CNN layers in Keras

Constructing a layer with the `Conv2D` class requires us to specify the number of output filters (which is equivalent to the number of output feature maps) and kernel sizes.

In addition, there are optional parameters that we can use to configure a convolutional layer. The most commonly used ones are the strides (with a default value of 1 in both *x*, *y* dimensions) and padding, which could be same or valid. Additional configuration parameters are listed in the official documentation: `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Conv2D`.

It is worth mentioning that usually, when we read an image, the default dimension for the channels is the last dimension of the tensor array. This is called the "NHWC" format, where *N* stands for the number of images within the batch, *H* and *W* stand for height and width, respectively, and *C* stands for channels.

Note that the `Conv2D` class assumes that inputs are in the NHWC format by default. (Other tools, such as PyTorch, use an NCHW format.) However, if you come across some data whose channels are placed at the first dimension (the first dimension after the batch dimension, or second dimension considering the batch dimension), you would need to swap the axes in your data to move the channels to the last dimension. Or, an alternative way to work with an NCHW-formatted input is to set `data_format="channels_first"`. After the layer is constructed, it can be called by providing a four-dimensional tensor, with the first dimension reserved for a batch of examples; depending on the `data_format` argument, either the second or the fourth dimension corresponds to the channel; and the other two dimensions are the spatial dimensions.

As shown in the architecture of the CNN model that we want to build, each convolution layer is followed by a pooling layer for subsampling (reducing the size of feature maps). The `MaxPool2D` and `AvgPool2D` classes construct the max-pooling and average-pooling layers, respectively. The argument `pool_size` determines the size of the window (or neighborhood) that will be used to compute the max or mean operations. Furthermore, the `strides` parameter can be used to configure the pooling layer, as we discussed earlier.

Finally, the `Dropout` class will construct the dropout layer for regularization, with the argument `rate` used to determine the probability of dropping the input units during the training. When calling this layer, its behavior can be controlled via an argument named `training`, to specify whether this call will be made during training or during the inference.

Constructing a CNN in Keras

Now that you have learned about these classes, we can construct the CNN model that was shown in the previous figure. In the following code, we will use the `Sequential` class and add the convolution and pooling layers:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Conv2D(
...     filters=32, kernel_size=(5, 5),
...     strides=(1, 1), padding='same',
...     data_format='channels_last',
...     name='conv_1', activation='relu'))
>>> model.add(tf.keras.layers.MaxPool2D(
...     pool_size=(2, 2), name='pool_1'))

>>> model.add(tf.keras.layers.Conv2D(
...     filters=64, kernel_size=(5, 5),
...     strides=(1, 1), padding='same',
...     name='conv_2', activation='relu'))
>>> model.add(tf.keras.layers.MaxPool2D(
...     pool_size=(2, 2), name='pool_2'))
```

So far, we have added two convolution layers to the model. For each convolutional layer, we used a kernel of size 5 × 5 and `'same'` padding. As discussed earlier, using `padding='same'` preserves the spatial dimensions (vertical and horizontal dimensions) of the feature maps such that the inputs and outputs have the same height and width (and the number of channels may only differ in terms of the number of filters used). The max-pooling layers with pooling size 2 × 2 and strides of 2 will reduce the spatial dimensions by half. (Note that if the `strides` parameter is not specified in `MaxPool2D`, by default, it is set equal to the pooling size.)

While we can calculate the size of the feature maps at this stage manually, the Keras API provides a convenient method to compute this for us:

```
>>> model.compute_output_shape(input_shape=(16, 28, 28, 1))
TensorShape([16, 7, 7, 64])
```

By providing the input shape as a tuple specified in this example, the method `compute_output_shape` calculated the output to have a shape (16, 7, 7, 64), indicating feature maps with 64 channels and a spatial size of 7 × 7. The first dimension corresponds to the batch dimension, for which we used 16 arbitrarily. We could have used `None` instead, that is, `input_shape=(None, 28, 28, 1)`.

The next layer that we want to add is a dense (or fully connected) layer for implementing a classifier on top of our convolutional and pooling layers. The input to this layer must have rank 2, that is, shape [*batchsize* × *input_units*]. Thus, we need to flatten the output of the previous layers to meet this requirement for the dense layer:

```
>>> model.add(tf.keras.layers.Flatten())
>>> model.compute_output_shape(input_shape=(16, 28, 28, 1))
TensorShape([16, 3136])
```

As the result of compute_output_shape indicates, the input dimensions for the dense layer are correctly set up. Next, we will add two dense layers with a dropout layer in between:

```
>>> model.add(tf.keras.layers.Dense(
...     units=1024, name='fc_1',
...     activation='relu'))

>>> model.add(tf.keras.layers.Dropout(
...     rate=0.5))

>>> model.add(tf.keras.layers.Dense(
...     units=10, name='fc_2',
...     activation='softmax'))
```

The last fully connected layer, named 'fc_2', has 10 output units for the 10 class labels in the MNIST dataset. Also, we use the softmax activation to obtain the class-membership probabilities of each input example, assuming that the classes are mutually exclusive, so the probabilities for each example sum to 1. (This means that a training example can belong to only one class.) Based on what we discussed in the section *Loss functions for classification*, which loss should we use here? Remember that for a multiclass classification with integer (sparse) labels (as opposed to one-hot encoded labels), we use SparseCategoricalCrossentropy. The following code will call the build() method for late variable creation and compile the model:

```
>>> tf.random.set_seed(1)
>>> model.build(input_shape=(None, 28, 28, 1))
>>> model.compile(
...     optimizer=tf.keras.optimizers.Adam(),
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(),
...     metrics=['accuracy'])
```

The Adam optimizer

Note that in this implementation, we used the tf.keras.optimizers.Adam() class for training the CNN model. The Adam optimizer is a robust, gradient-based optimization method suited to nonconvex optimization and machine learning problems. Two popular optimization methods inspired Adam: RMSProp and AdaGrad.

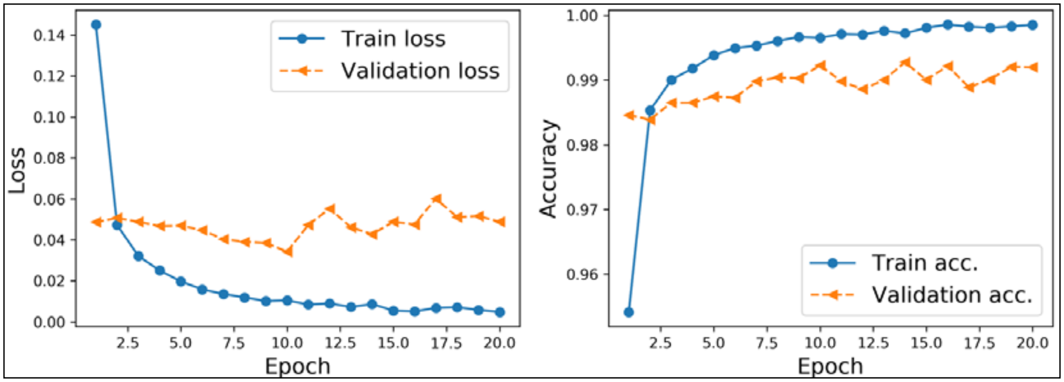
The key advantage of Adam is in the choice of update step size derived from the running average of gradient moments. Please feel free to read more about the Adam optimizer in the manuscript, *Adam: A Method for Stochastic Optimization*, *Diederik P. Kingma* and *Jimmy Lei Ba*, 2014. The article is freely available at https://arxiv.org/abs/1412.6980.

As you already know, we can train the model by calling the `fit()` method. Note that using the designated methods for training and evaluation (like `evaluate()` and `predict()`) will automatically set the mode for the dropout layer and rescale the hidden units appropriately so that we do not have to worry about that at all. Next, we will train this CNN model and use the validation dataset that we created for monitoring the learning progress:

```
>>> history = model.fit(mnist_train, epochs=NUM_EPOCHS,
...                     validation_data=mnist_valid,
...                     shuffle=True)

Epoch 1/20
782/782 [==============================] - 35s 45ms/step - loss:
0.1450 - accuracy: 0.8882 - val_loss: 0.0000e+00 - val_accuracy:
0.0000e+00
Epoch 2/20
782/782 [==============================] - 34s 43ms/step - loss:
0.0472 - accuracy: 0.9833 - val_loss: 0.0507 - val_accuracy: 0.9839
..
Epoch 20/20
782/782 [==============================] - 34s 44ms/step - loss:
0.0047 - accuracy: 0.9985 - val_loss: 0.0488 - val_accuracy: 0.9920
```

Once the 20 epochs of training are finished, we can visualize the learning curves:

```
>>> import matplotlib.pyplot as plt

>>> hist = history.history
>>> x_arr = np.arange(len(hist['loss'])) + 1

>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
>>> ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
>>> ax.legend(fontsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
>>> ax.plot(x_arr, hist['val_accuracy'], '--<',
...         label='Validation acc.')
>>> ax.legend(fontsize=15)
>>> plt.show()
```



As you already know from the two previous chapters, evaluating the trained model on the test dataset can be done by calling the `.evaluate()` method:

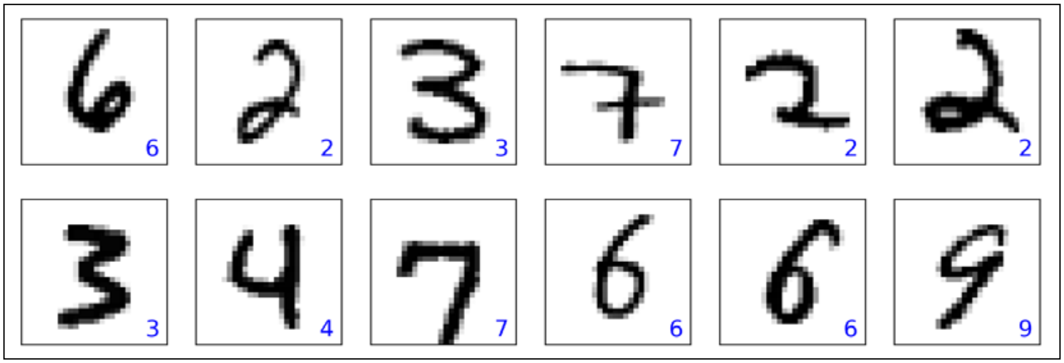
```
>>> test_results = model.evaluate(mnist_test.batch(20))
>>> print('Test Acc.: {:.2f}\%'.format(test_results[1]*100))
Test Acc.: 99.39%
```

The CNN model achieves an accuracy of 99.39%. Remember that in *Chapter 14, Going Deeper – The Mechanics of TensorFlow*, we got approximately 90% accuracy using the Estimator `DNNClassifier`.

Finally, we can get the prediction results in the form of class-membership probabilities and convert them to predicted labels by using the `tf.argmax` function to find the element with the maximum probability. We will do this for a batch of 12 examples and visualize the input and predicted labels:

```
>>> batch_test = next(iter(mnist_test.batch(12)))

>>> preds = model(batch_test[0])
>>> tf.print(preds.shape)
TensorShape([12, 10])

>>> preds = tf.argmax(preds, axis=1)
>>> print(preds)
tf.Tensor([6 2 3 7 2 2 3 4 7 6 6 9], shape=(12,), dtype=int64)


>>> fig = plt.figure(figsize=(12, 4))
>>> for i in range(12):
...     ax = fig.add_subplot(2, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     img = batch_test[0][i, :, :, 0]
...     ax.imshow(img, cmap='gray_r')
...     ax.text(0.9, 0.1, '{}'.format(preds[i]),
...             size=15, color='blue',
...             horizontalalignment='center',
...             verticalalignment='center',
...             transform=ax.transAxes)

>>> plt.show()
```

The following figure shows the handwritten inputs and their predicted labels:



In this set of plotted examples, all the predicted labels are correct.

We leave the task of showing some of the misclassified digits, as we did in *Chapter 12, Implementing a Multilayer Artificial Neural Network from Scratch*, as an exercise for the reader.

Gender classification from face images using a CNN

In this section, we are going to implement a CNN for gender classification from face images using the CelebA dataset. As you already saw in *Chapter 13, Parallelizing Neural Network Training with TensorFlow,* the CelebA dataset contains 202,599 face images of celebrities. In addition, 40 binary facial attributes are available for each image, including gender (male or female) and age (young or old).

Based on what you have learned so far, the goal of this section is to build and train a CNN model for predicting the gender attribute from these face images. Here, for simplicity, we will only be using a small portion of the training data (16,000 training examples) to speed up the training process. However, in order to improve the generalization performance and reduce overfitting on such a small dataset, we will use a technique called **data augmentation**.

Loading the CelebA dataset

First, let's load the data similarly to how we did in the previous section for the MNIST dataset. CelebA data comes in three partitions: a training dataset, a validation dataset, and a test dataset. Next, we will implement a simple function to count the number of examples in each partition:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> celeba_bldr = tfds.builder('celeb_a')
>>> celeba_bldr.download_and_prepare()
>>> celeba = celeba_bldr.as_dataset(shuffle_files=False)

>>> celeba_train = celeba['train']
>>> celeba_valid = celeba['validation']
>>> celeba_test = celeba['test']
>>>
>>> def count_items(ds):
...     n = 0
...     for _ in ds:
...         n += 1
...     return n

>>> print('Train set:  {}'.format(count_items(celeba_train)))
Train set:  162770

>>> print('Validation: {}'.format(count_items(celeba_valid)))
Validation: 19867

>>> print('Test set:   {}'.format(count_items(celeba_test)))
Test set:   19962
```

So, instead of using all the available training and validation data, we will take a subset of 16,000 training examples and 1,000 examples for validation, as follows:

```
>>> celeba_train = celeba_train.take(16000)
>>> celeba_valid = celeba_valid.take(1000)
>>> print('Train set:  {}'.format(count_items(celeba_train)))
Train set:  16000

>>> print('Validation: {}'.format(count_items(celeba_valid)))
Validation: 1000
```

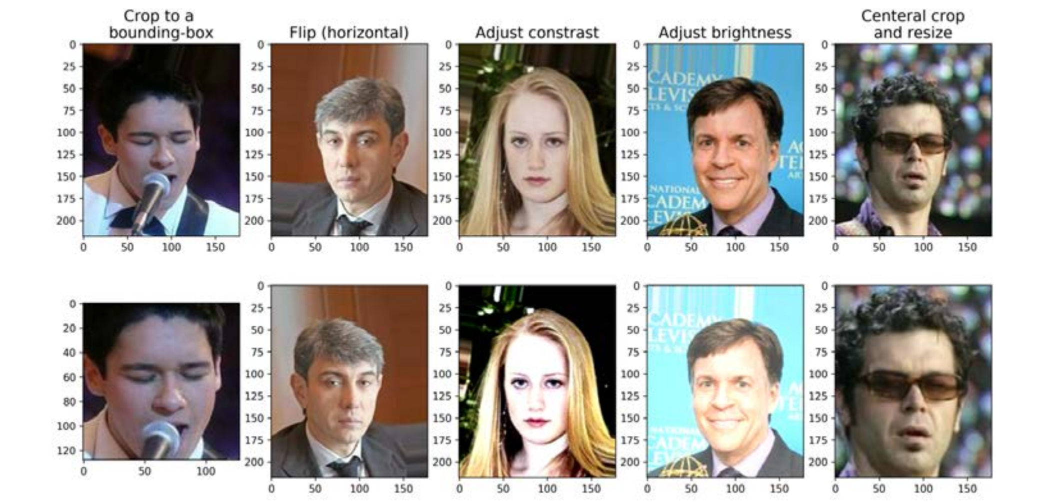
It is important to note that if the `shuffle_files` argument in `celeba_bldr.as_dataset()` was not set to `False`, we would still see 16,000 examples in the training dataset and 1,000 examples for the validation dataset. However, at each iteration, it would reshuffle the training data and take a new set of 16,000 examples. This would defeat the purpose, as our goal here is to intentionally train our model with a small dataset. Next, we will discuss data augmentation as a technique for boosting the performance of deep NNs.

Image transformation and data augmentation

Data augmentation summarizes a broad set of techniques for dealing with cases where the training data is limited. For instance, certain data augmentation techniques allow us to modify or even artificially synthesize more data and thereby boost the performance of a machine or deep learning model by reducing overfitting. While data augmentation is not only for image data, there are a set of transformations uniquely applicable to image data, such as cropping parts of an image, flipping, changing the contrast, brightness, and saturation. Let's see some of these transformations that are available via the `tf.image` module. In the following code block, we will first get five examples from the `celeba_train` dataset and apply five different types of transformation: 1) cropping an image to a bounding box, 2) flipping an image horizontally, 3) adjusting the contrast, 4) adjusting the brightness, and 5) center-cropping an image and resizing the resulting image back to its original size, (218, 178). In the following code, we will visualize the results of these transformations, showing each one in a separate column for comparison:

```
>>> import matplotlib.pyplot as plt
>>> # take 5 examples
>>> examples = []
>>> for example in celeba_train.take(5):
...     examples.append(example['image'])

>>> fig = plt.figure(figsize=(16, 8.5))


>>> ## Column 1: cropping to a bounding-box
>>> ax = fig.add_subplot(2, 5, 1)
>>> ax.set_title('Crop to a \nbounding-box', size=15)
>>> ax.imshow(examples[0])
>>> ax = fig.add_subplot(2, 5, 6)
>>> img_cropped = tf.image.crop_to_bounding_box(
...     examples[0], 50, 20, 128, 128)
>>> ax.imshow(img_cropped)

>>> ## Column 2: flipping (horizontally)
>>> ax = fig.add_subplot(2, 5, 2)
>>> ax.set_title('Flip (horizontal)', size=15)
>>> ax.imshow(examples[1])
>>> ax = fig.add_subplot(2, 5, 7)
>>> img_flipped = tf.image.flip_left_right(examples[1])
>>> ax.imshow(img_flipped)

>>> ## Column 3: adjust contrast
>>> ax = fig.add_subplot(2, 5, 3)
>>> ax.set_title('Adjust constrast', size=15)
>>> ax.imshow(examples[2])
>>> ax = fig.add_subplot(2, 5, 8)
>>> img_adj_contrast = tf.image.adjust_contrast(
...     examples[2], contrast_factor=2)
>>> ax.imshow(img_adj_contrast)

>>> ## Column 4: adjust brightness
>>> ax = fig.add_subplot(2, 5, 4)
>>> ax.set_title('Adjust brightness', size=15)
>>> ax.imshow(examples[3])
>>> ax = fig.add_subplot(2, 5, 9)
>>> img_adj_brightness = tf.image.adjust_brightness(
...     examples[3], delta=0.3)
>>> ax.imshow(img_adj_brightness)

>>> ## Column 5: cropping from image center
>>> ax = fig.add_subplot(2, 5, 5)
>>> ax.set_title('Centeral crop\nand resize', size=15)
>>> ax.imshow(examples[4])
>>> ax = fig.add_subplot(2, 5, 10)
>>> img_center_crop = tf.image.central_crop(
...     examples[4], 0.7)
>>> img_resized = tf.image.resize(
...     img_center_crop, size=(218, 178))
>>> ax.imshow(img_resized.numpy().astype('uint8'))

>>> plt.show()
```

The following figure shows the results:



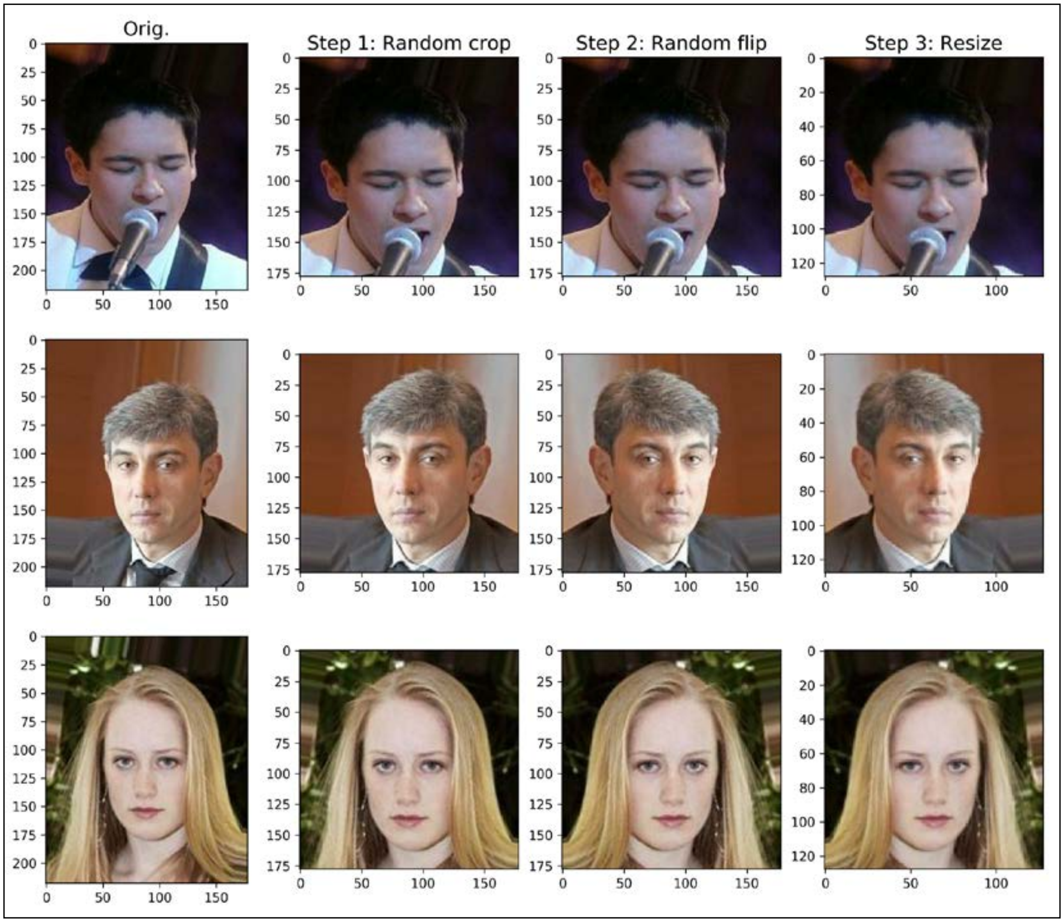
In the previous figure, the original images are shown in the first row and their transformed version in the second row. Note that for the first transformation (leftmost column), the bounding box is specified by four numbers: the coordinate of the upper-left corner of the bounding box (here *x*=20, *y*=50), and the width and height of the box (width=128, height=128). Also note that the origin (the coordinates at the location denoted as (0, 0)) for images loaded by TensorFlow (as well as other packages such as `imageio`) is the upper-left corner of the image.

The transformations in the previous code block are deterministic. However, all such transformations can also be randomized, which is recommended for data augmentation during model training. For example, a random bounding box (where the coordinates of the upper-left corner are selected randomly) can be cropped from an image, an image can be randomly flipped along either the horizontal or vertical axes with a probability of 0.5, or the contrast of an image can be changed randomly, where the `contrast_factor` is selected at random, but with uniform distribution, from a range of values. In addition, we can create a pipeline of these transformations.

For example, we can first randomly crop an image, then flip it randomly, and finally, resize it to the desired size. The code is as follows (since we have random elements, we set the random seed for reproducibility):

```
>>> tf.random.set_seed(1)

>>> fig = plt.figure(figsize=(14, 12))

>>> for i,example in enumerate(celeba_train.take(3)):
...     image = example['image']
...
...     ax = fig.add_subplot(3, 4, i*4+1)
...     ax.imshow(image)
...     if i == 0:
...         ax.set_title('Orig', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+2)
...     img_crop = tf.image.random_crop(image, size=(178, 178, 3))
...     ax.imshow(img_crop)
...     if i == 0:
...         ax.set_title('Step 1: Random crop', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+3)
...     img_flip = tf.image.random_flip_left_right(img_crop)
...     ax.imshow(tf.cast(img_flip, tf.uint8))
...     if i == 0:
...         ax.set_title('Step 2: Random flip', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+4)
...     img_resize = tf.image.resize(img_flip, size=(128, 128))
...     ax.imshow(tf.cast(img_resize, tf.uint8))
...     if i == 0:
...         ax.set_title('Step 3: Resize', size=15)

>>> plt.show()
```

The following figure shows random transformations on three example images:



Note that each time we iterate through these three examples, we get slightly different images due to random transformations.

For convenience, we can define a wrapper function to use this pipeline for data augmentation during model training. In the following code, we will define the function `preprocess()`, which will receive a dictionary containing the keys `'image'` and `'attributes'`. The function will return a tuple containing the transformed image and the label extracted from the dictionary of attributes.

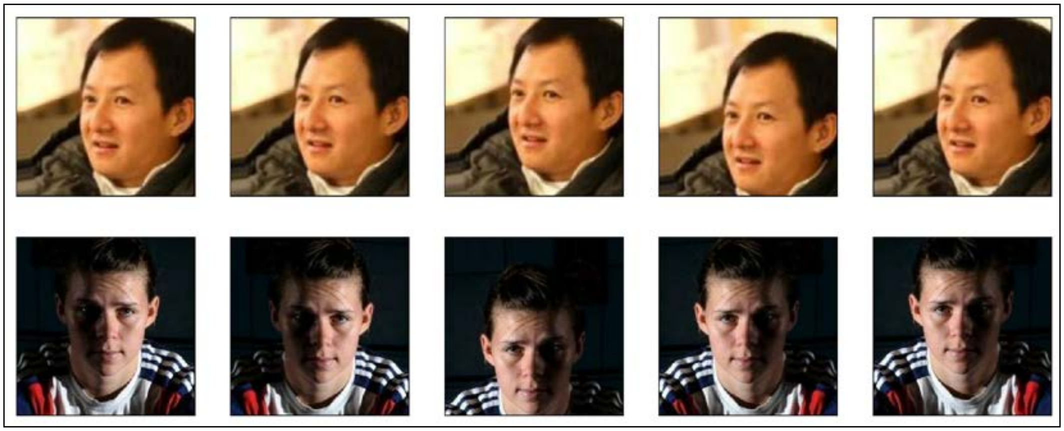
We will only apply data augmentation to the training examples, however, and not to the validation or test images. The code is as follows:

```
>>> def preprocess(example, size=(64, 64), mode='train'):
...     image = example['image']
...     label = example['attributes']['Male']
...     if mode == 'train':
...         image_cropped = tf.image.random_crop(
...             image, size=(178, 178, 3))
...         image_resized = tf.image.resize(
...             image_cropped, size=size)
...         image_flip = tf.image.random_flip_left_right(
...             image_resized)
...         return image_flip/255.0, tf.cast(label, tf.int32)
...     else: # use center- instead of
...           # random-crops for non-training data
...         image_cropped = tf.image.crop_to_bounding_box(
...             image, offset_height=20, offset_width=0,
...             target_height=178, target_width=178)
...         image_resized = tf.image.resize(
...             image_cropped, size=size)
...         return image_resized/255.0, tf.cast(label, tf.int32)
```

Now, to see data augmentation in action, let's create a small subset of the training dataset, apply this function to it, and iterate over the dataset five times:

```
>>> tf.random.set_seed(1)

>>> ds = celeba_train.shuffle(1000, reshuffle_each_iteration=False)
>>> ds = ds.take(2).repeat(5)

>>> ds = ds.map(lambda x:preprocess(x, size=(178, 178), mode='train'))

>>> fig = plt.figure(figsize=(15, 6))
>>> for j,example in enumerate(ds):
...     ax = fig.add_subplot(2, 5, j//2+(j%2)*5+1)
...     ax.set_xticks([])
...     ax.set_yticks([])
...     ax.imshow(example[0])
>>> plt.show()
```

This figure shows the five resulting transformations for data augmentation on two example images:



Next, we will apply this preprocessing function to our training and validation datasets. We will use an image size of `(64, 64)`. Furthermore, we will specify `mode='train'` when working with training data and use `mode='eval'` for the validation data so that the random elements of the data augmentation pipeline will be applied only to the training data:

```
>>> import numpy as np

>>> BATCH_SIZE = 32
>>> BUFFER_SIZE = 1000
>>> IMAGE_SIZE = (64, 64)
>>> steps_per_epoch = np.ceil(16000/BATCH_SIZE)

>>> ds_train = celeba_train.map(
...     lambda x: preprocess(x, size=IMAGE_SIZE, mode='train'))
>>> ds_train = ds_train.shuffle(buffer_size=BUFFER_SIZE).repeat()
>>> ds_train = ds_train.batch(BATCH_SIZE)

>>> ds_valid = celeba_valid.map(
...     lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval'))
>>> ds_valid = ds_valid.batch(BATCH_SIZE)
```

Training a CNN gender classifier

By now, building a model with TensorFlow's Keras API and training it should be straightforward. The design of our CNN is as follows: the CNN model receives input images of size 64 × 64 × 3 (the images have three color channels, using `'channels_last'`).

The input data goes through four convolutional layers to make 32, 64, 128, and 256 feature maps using filters with kernel a size of 3 × 3. The first three convolution layers are followed by max-pooling, $P_{2\times2}$. Two dropout layers are also included for regularization:

```
>>> model = tf.keras.Sequential([
...     tf.keras.layers.Conv2D(
...         32, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         64, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         128, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...
...     tf.keras.layers.Conv2D(
...         256, (3, 3), padding='same', activation='relu')
>>>     ])
```

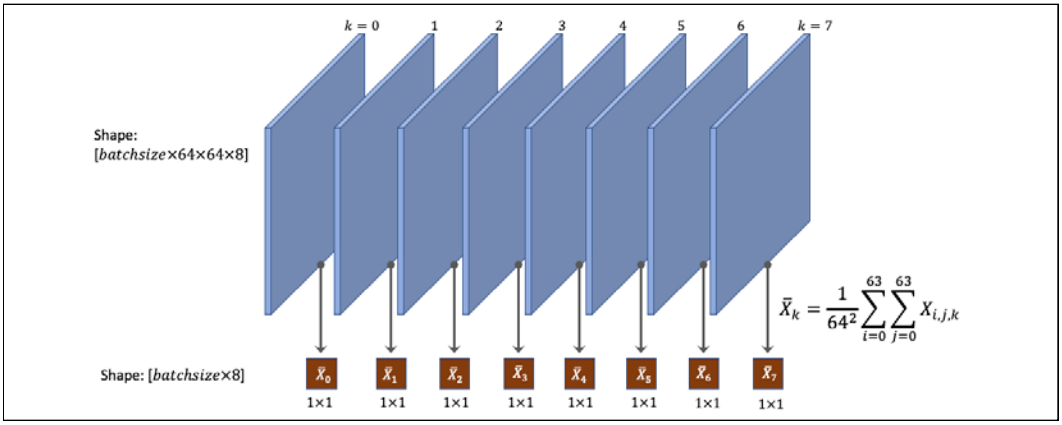
Let's see the shape of the output feature maps after applying these layers:

```
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 8, 8, 256])
```

There are 256 feature maps (or channels) of size 8 × 8. Now, we can add a fully connected layer to get to the output layer with a single unit. If we reshape (flatten) the feature maps, the number of input units to this fully connected layer will be $8 \times 8 \times 256 = 16{,}384$. Alternatively, let's consider a new layer, called *global average-pooling*, which computes the average of each feature map separately, thereby reducing the hidden units to 256. We can then add a fully connected layer. Although we have not discussed global average-pooling explicitly, it is conceptually very similar to other pooling layers. Global average-pooling can be viewed, in fact, as a special case of average-pooling when the pooling size is equal to the size of the input feature maps.

To understand this, consider the following figure showing an example of input feature maps of shape $[batchsize \times 64 \times 64 \times 8]$. The channels are numbered $k = 0, 1, \dots, 7$. The global average-pooling operation calculates the average of each channel so that the output will have shape $[batchsize \times 8]$. (Note: `GlobalAveragePooling2D` in the Keras API will automatically squeeze the output.

Without squeezing the output, the shape would be [$batchsize \times 1 \times 1 \times 8$], as the global average-pooling would reduce the spatial dimension of 64 × 64 to a 1 × 1):



Therefore, given that, in our case, the shape of the feature maps prior to this layer is [$batchsize \times 8 \times 8 \times 256$], we expect to get 256 units as output, that is, the shape of the output will be [$batchsize \times 256$]. Let's add this layer and recompute the output shape to verify that this is true:

```
>>> model.add(tf.keras.layers.GlobalAveragePooling2D())
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 256])
```

Finally, we can add a fully connected (dense) layer to get a single output unit. In this case, we can specify the activation function to be `'sigmoid'` or just use `activation=None`, so that the model will output the logits (instead of class-membership probabilities), which is preferred for model training in TensorFlow and Keras due to numerical stability, as discussed earlier:

```
>>> model.add(tf.keras.layers.Dense(1, activation=None))
>>> tf.random.set_seed(1)
>>> model.build(input_shape=(None, 64, 64, 3))
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              multiple                  896
_________________________________________________________________
max_pooling2d (MaxPooling2D) multiple                  0
_________________________________________________________________
dropout (Dropout)            multiple                  0
```

```
_________________________________________________________________
conv2d_1 (Conv2D)            multiple                  18496     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 multiple                  0         
_________________________________________________________________
dropout_1 (Dropout)          multiple                  0         
_________________________________________________________________
conv2d_2 (Conv2D)            multiple                  73856     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 multiple                  0         
_________________________________________________________________
conv2d_3 (Conv2D)            multiple                  295168    
_________________________________________________________________
global_average_pooling2d (Gl multiple                  0         
_________________________________________________________________
dense (Dense)                multiple                  257       
=================================================================
Total params: 388,673
Trainable params: 388,673
Non-trainable params: 0
_________________________________________________________________
```

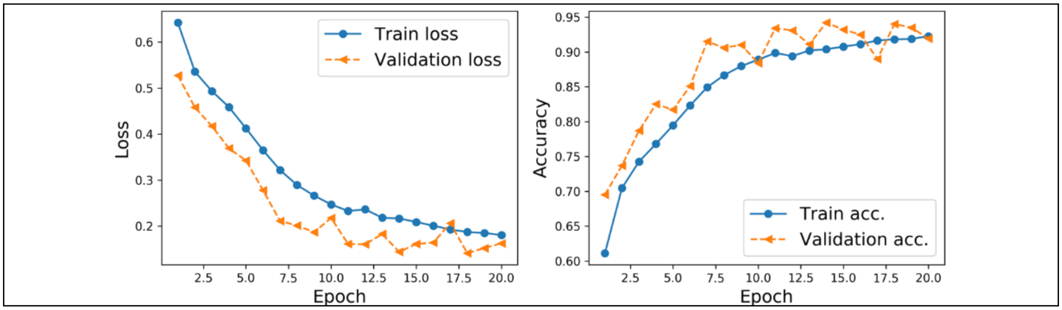
The next step is to compile the model and, at this time, we have to decide what loss function to use. We have a binary classification with a single output unit, so that means we should use `BinaryCrossentropy`. In addition, since our last layer does not apply sigmoid activation (we used `activation=None`), the outputs of our model are logits, not probabilities. Therefore, we will also specify `from_logits=True` in `BinaryCrossentropy` so that the loss function applies the sigmoid function internally, which is, due to the underlying code, more efficient than doing it manually. The code for compiling and training the model is as follows:

```
>>> model.compile(optimizer=tf.keras.optimizers.Adam(),
...     loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
...               metrics=['accuracy'])

>>> history = model.fit(ds_train, validation_data=ds_valid,
...                     epochs=20,
...                     steps_per_epoch=steps_per_epoch)
```

Let's now visualize the learning curve and compare the training and validation loss and accuracies after each epoch:

```
>>> hist = history.history
>>> x_arr = np.arange(len(hist['loss'])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
>>> ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.set_ylabel('Loss', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
>>> ax.plot(x_arr, hist['val_accuracy'], '--<',
...         label='Validation acc.')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.set_ylabel('Accuracy', size=15)
>>> plt.show()
```

The following figure shows the training and validation losses and accuracies:



As you can see from the learning curves, the losses for the training and validation have not converged to a plateau region. Based on this result, we could have continued training for a few more epochs. Using the `fit()` method, we can continue training for an additional 10 epochs as follows:

```
>>> history = model.fit(ds_train, validation_data=ds_valid,
...                     epochs=30, initial_epoch=20,
...                     steps_per_epoch=steps_per_epoch)
```

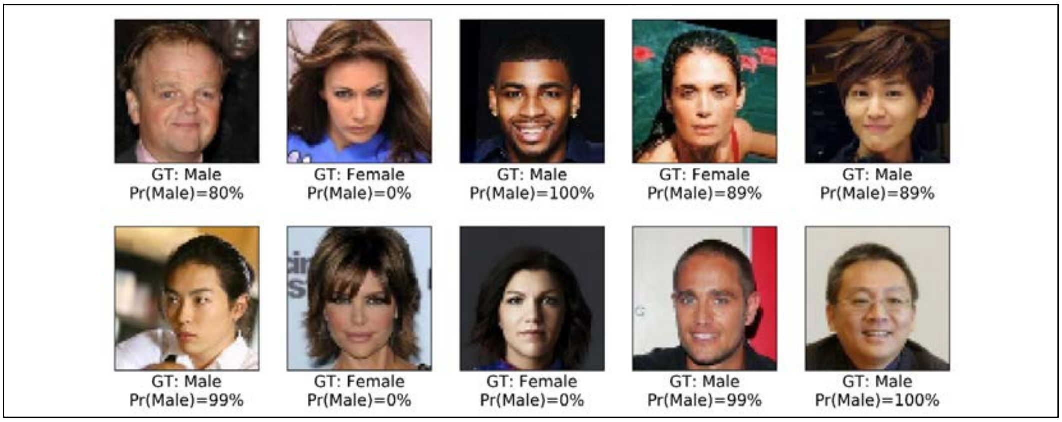
Once we are happy with the learning curves, we can evaluate the model on the hold-out test dataset:

```
>>> ds_test = celeba_test.map(
...   lambda x:preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
>>> test_results = model.evaluate(ds_test)
>>> print('Test Acc: {:.2f}%'.format(test_results[1]*100))
Test Acc: 94.75%
```

Finally, we already know how to get the prediction results on some test examples using `model.predict()`. However, remember that the model outputs the logits, not probabilities. If we are interested in the class-membership probabilities for this binary problem with a single output unit, we can use the `tf.sigmoid` function to compute the probability of class 1. (In the case of a multiclass problem, we would use `tf.math.softmax`.) In the following code, we will take a small subset of 10 examples from our pre-processed test dataset (`ds_test`) and run `model.predict()` to get the logits. Then, we will compute the probabilities of each example being from class 1 (which corresponds to *male* based on the labels provided in CelebA) and visualize the examples along with their ground truth label and the predicted probabilities. Notice that we first apply `unbatch()` to the `ds_test` dataset before taking 10 examples; otherwise, the `take()` method would return 10 batches of size 32, instead of 10 individual examples:

```
>>> ds = ds_test.unbatch().take(10)

>>> pred_logits = model.predict(ds.batch(10))
>>> probas = tf.sigmoid(pred_logits)
>>> probas = probas.numpy().flatten()*100

>>> fig = plt.figure(figsize=(15, 7))
>>> for j,example in enumerate(ds):
...     ax = fig.add_subplot(2, 5, j+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0])
...     if example[1].numpy() == 1:
...         label='M'
...     else:
...         label = 'F'
...     ax.text(
...         0.5, -0.15, 'GT: {:s}\nPr(Male)={:.0f}%'
...         ''.format(label, probas[j]),
...         size=16,
...         horizontalalignment='center',
...         verticalalignment='center',
...         transform=ax.transAxes)
>>> plt.tight_layout()
>>> plt.show()
```

In the following figure, you can see 10 example images along with their ground truth labels and the probabilities that they belong to class 1, male:



The probabilities of class 1 (that is, *male* according to CelebA) are provided below each image. As you can see, our trained model made only one error on this set of 10 test examples.

As an optional exercise, you are encouraged to try using the entire training dataset instead of the small subset we created. Furthermore, you can change or modify the CNN architecture. For example, you can change the dropout probabilities and the number of filters in the different convolutional layers. Also, you could replace the global average-pooling with a dense layer. If you are using the entire training dataset with the CNN architecture we trained in this chapter, you should be able to achieve about 97-99% accuracy.

Summary

In this chapter, we learned about CNNs and their main components. We started with the convolution operation and looked at 1D and 2D implementations. Then, we covered another type of layer that is found in several common CNN architectures: the subsampling or so-called pooling layers. We primarily focused on the two most common forms of pooling: max-pooling and average-pooling.

Next, putting all these individual concepts together, we implemented deep CNNs using the TensorFlow Keras API. The first network we implemented was applied to the already familiar MNIST handwritten digit recognition problem.

Then, we implemented a second CNN on a more complex dataset consisting of face images and trained the CNN for gender classification. Along the way, you also learned about data augmentation and different transformations that we can apply to face images using the TensorFlow Dataset class.

In the next chapter, we will move on to **recurrent neural networks** (**RNNs**). RNNs are used for learning the structure of sequence data, and they have some fascinating applications, including language translation and image captioning.

16

Modeling Sequential Data Using Recurrent Neural Networks

In the previous chapter, we focused on **convolutional neural networks** (**CNNs**). We covered the building blocks of CNN architectures and how to implement deep CNNs in TensorFlow. Finally, you learned how to use CNNs for image classification. In this chapter, we will explore **recurrent neural networks** (**RNNs**) and see their application in modeling sequential data.

We will cover the following topics:

- Introducing sequential data
- RNNs for modeling sequences
- Long short-term memory (LSTM)
- Truncated backpropagation through time (TBPTT)
- Implementing a multilayer RNN for sequence modeling in TensorFlow
- Project one: RNN sentiment analysis of the IMDb movie review dataset
- Project two: RNN character-level language modeling with LSTM cells, using text data from Jules Verne's *The Mysterious Island*
- Using gradient clipping to avoid exploding gradients
- Introducing the *Transformer* model and understanding the *self-attention mechanism*

Introducing sequential data

Let's begin our discussion of RNNs by looking at the nature of sequential data, which is more commonly known as sequence data or **sequences**. We will take a look at the unique properties of sequences that make them different to other kinds of data. We will then see how we can represent sequential data and explore the various categories of models for sequential data, which are based on the input and output of a model. This will help us to explore the relationship between RNNs and sequences in this chapter.

Modeling sequential data – order matters

What makes sequences unique, compared to other types of data, is that elements in a sequence appear in a certain order and are not independent of each other. Typical machine learning algorithms for supervised learning assume that the input is **independent and identically distributed** (**IID**) data, which means that the training examples are *mutually independent* and have the same underlying distribution. In this regard, based on the mutual independence assumption, the order in which the training examples are given to the model is irrelevant. For example, if we have a sample consisting of n training examples, $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$, the order in which we use the data for training our machine learning algorithm does not matter. An example of this scenario would be the Iris dataset that we previously worked with. In the Iris dataset, each flower has been measured independently, and the measurements of one flower do not influence the measurements of another flower.

However, this assumption is not valid when we deal with sequences—by definition, order matters. Predicting the market value of a particular stock would be an example of this scenario. For instance, assume we have a sample of n training examples, where each training example represents the market value of a certain stock on a particular day. If our task is to predict the stock market value for the next three days, it would make sense to consider the previous stock prices in a date-sorted order to derive trends rather than utilize these training examples in a randomized order.

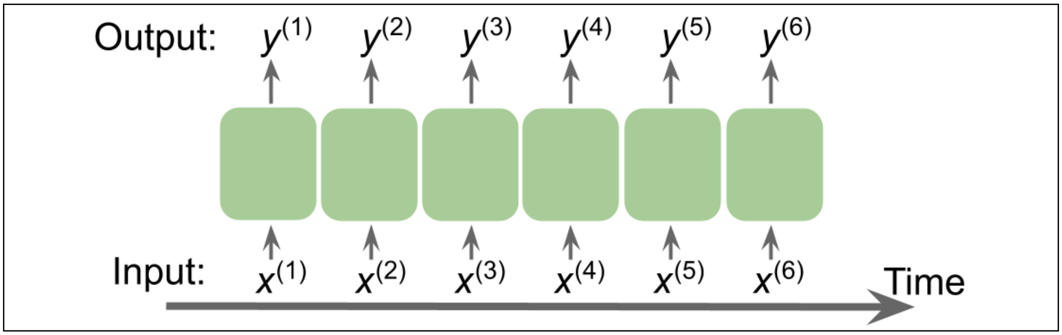
Sequential data versus time-series data

Time-series data is a special type of sequential data, where each example is associated with a dimension for time. In time-series data, samples are taken at successive time stamps, and therefore, the time dimension determines the order among the data points. For example, stock prices and voice or speech records are time-series data.

On the other hand, not all sequential data has the time dimension, for example, text data or DNA sequences, where the examples are ordered but they do not qualify as time-series data. As you will see, in this chapter, we will cover some examples of natural language processing (NLP) and text modeling that are not time-series data, but note that RNNs can also be used for time-series data.

Representing sequences

We've established that order among data points is important in sequential data, so we next need to find a way to leverage this ordering information in a machine learning model. Throughout this chapter, we will represent sequences as $\langle \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(T)} \rangle$. The superscript indices indicate the order of the instances, and the length of the sequence is *T*. For a sensible example of sequences, consider time-series data, where each example point, $x^{(t)}$, belongs to a particular time, *t*. The following figure shows an example of time-series data where both the input features (***x***'s) and the target labels (***y***'s) naturally follow the order according to their time axis; therefore, both the ***x***'s and ***y***'s are sequences:

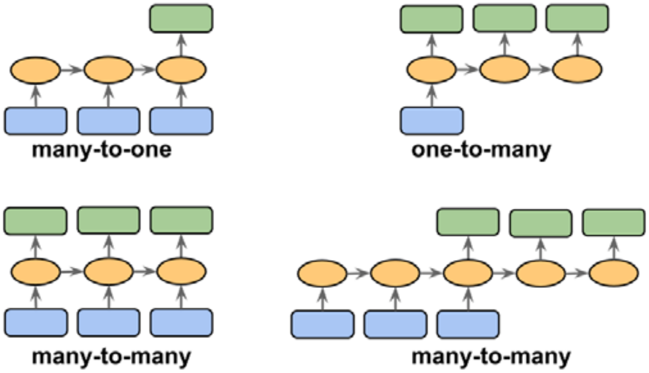

Output:
$y^{(1)}$ $y^{(2)}$ $y^{(3)}$ $y^{(4)}$ $y^{(5)}$ $y^{(6)}$
Input:
$x^{(1)}$ $x^{(2)}$ $x^{(3)}$ $x^{(4)}$ $x^{(5)}$ $x^{(6)}$
Time

As we have already mentioned, the standard neural network (NN) models that we have covered so far, such as multilayer perceptron (MLP) and CNNs for image data, assume that the training examples are independent of each other and thus do not incorporate *ordering information*. We can say that such models do not have a *memory* of previously seen training examples. For instance, the samples are passed through the feedforward and backpropagation steps, and the weights are updated independently of the order in which the training examples are processed.

RNNs, by contrast, are designed for modeling sequences and are capable of remembering past information and processing new events accordingly, which is a clear advantage when working with sequence data.

The different categories of sequence modeling

Sequence modeling has many fascinating applications, such as language translation (for example, translating text from English to German), image captioning, and text generation. However, in order to choose an appropriate architecture and approach, we have to understand and be able to distinguish between these different sequence modeling tasks. The following figure, based on the explanations in the excellent article *The Unreasonable Effectiveness of Recurrent Neural Networks*, by *Andrej Karpathy* (`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`), summarizes the most common sequence modeling tasks, which depend on the relationship categories of input and output data:



Let's discuss the different relationship categories between input and output data, which were depicted in the previous figure, in more detail. If neither the input nor output data represents sequences, then we are dealing with standard data, and we could simply use a multilayer perceptron (or another classification model previously covered in this book) to model such data. However, if either the input or output is a sequence, the modeling task likely falls into one of these categories:

- **Many-to-one**: The input data is a sequence, but the output is a fixed-size vector or scalar, not a sequence. For example, in sentiment analysis, the input is text-based (for example, a movie review) and the output is a class label (for example, a label denoting whether a reviewer liked the movie).
- **One-to-many**: The input data is in standard format and not a sequence, but the output is a sequence. An example of this category is image captioning—the input is an image and the output is an English phrase summarizing the content of that image.

- **Many-to-many**: Both the input and output arrays are sequences. This category can be further divided based on whether the input and output are synchronized. An example of a synchronized many-to-many modeling task is video classification, where each frame in a video is labeled. An example of a *delayed* many-to-many modeling task would be translating one language into another. For instance, an entire English sentence must be read and processed by a machine before its translation into German is produced.

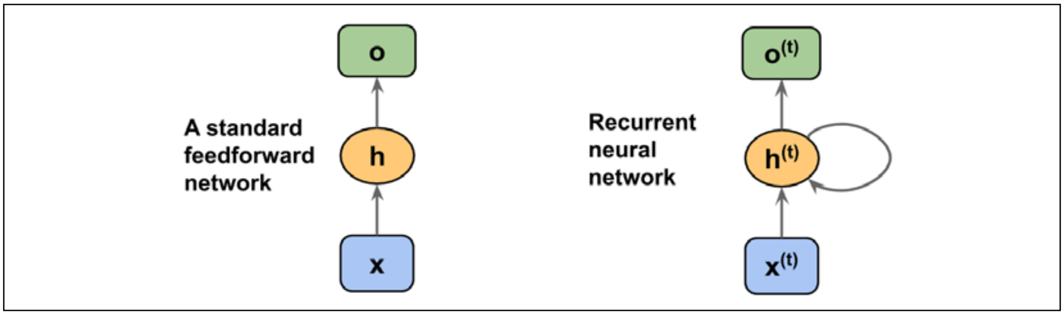
Now, after summarizing the three broad categories of sequence modeling, we can move forward to discussing the structure of an RNN.

RNNs for modeling sequences

In this section, before we start implementing RNNs in TensorFlow, we will discuss the main concepts of RNNs. We will begin by looking at the typical structure of an RNN, which includes a recursive component to model sequence data. Then, we will examine how the neuron activations are computed in a typical RNN. This will create a context for us to discuss the common challenges in training RNNs, and we will then discuss solutions to these challenges, such as LSTM and gated recurrent units (GRUs).

Understanding the RNN looping mechanism

Let's start with the architecture of an RNN. The following figure shows a standard feedforward NN and an RNN side by side for comparison:



Both of these networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer (***x***), hidden layer (***h***), and output layer (***o***) are vectors that contain many units.

Determining the type of output from an RNN

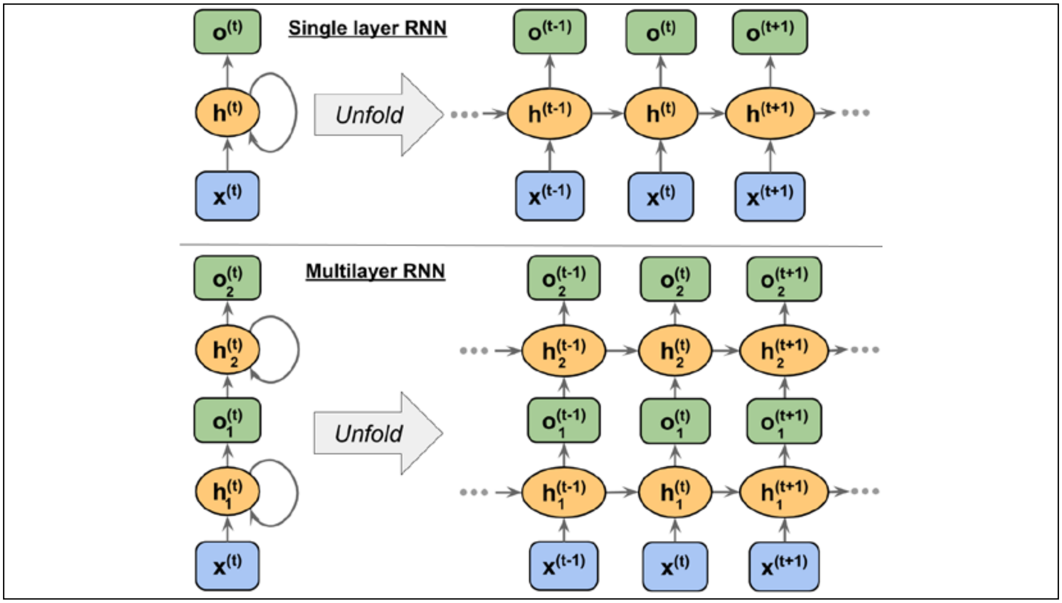
This generic RNN architecture could correspond to the two sequence modeling categories where the input is a sequence. Typically, a recurrent layer can return a sequence as output, $\langle \boldsymbol{o}^{(0)}, \boldsymbol{o}^{(1)}, \dots, \boldsymbol{o}^{(T)} \rangle$, or simply return the last output (at $t = T$, that is, $\boldsymbol{o}^{(T)}$). Thus, it could be either many-to-many, or it could be many-to-one if, for example, we only use the last element, $\boldsymbol{o}^{(T)}$, as the final output.

As you will see later, in the TensorFlow Keras API, the behavior of a recurrent layer with respect to returning a sequence as output or simply using the last output can be specified by setting the argument `return_sequences` to `True` or `False`, respectively.

In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step.

The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general RNN architecture got its name.

Similar to multilayer perceptrons, RNNs can consist of multiple hidden layers. Note that it's a common convention to refer to RNNs with one hidden layer as a *single-layer RNN*, which is not to be confused with single-layer NNs without a hidden layer, such as Adaline or logistic regression. The following figure illustrates an RNN with one hidden layer (top) and an RNN with two hidden layers (bottom):


Single layer RNN
$o^{(t)}$
$h^{(t)}$
$x^{(t)}$
Unfold
$o^{(t-1)}$ $o^{(t)}$ $o^{(t+1)}$
$h^{(t-1)}$ $h^{(t)}$ $h^{(t+1)}$
$x^{(t-1)}$ $x^{(t)}$ $x^{(t+1)}$
Multilayer RNN
$o_2^{(t)}$
$h_2^{(t)}$
$o_1^{(t)}$
$h_1^{(t)}$
$x^{(t)}$
Unfold
$o_2^{(t-1)}$ $o_2^{(t)}$ $o_2^{(t+1)}$
$h_2^{(t-1)}$ $h_2^{(t)}$ $h_2^{(t+1)}$
$o_1^{(t-1)}$ $o_1^{(t)}$ $o_1^{(t+1)}$
$h_1^{(t-1)}$ $h_1^{(t)}$ $h_1^{(t+1)}$
$x^{(t-1)}$ $x^{(t)}$ $x^{(t+1)}$

In order to examine the architecture of RNNs and the flow of information, a compact representation with a recurrent edge can be unfolded, which you can see in the preceding figure.

As we know, each hidden unit in a standard NN receives only one input—the net preactivation associated with the input layer. In contrast, each hidden unit in an RNN receives two *distinct* sets of input—the preactivation from the input layer and the activation of the same hidden layer from the previous time step, *t* – 1.

At the first time step, *t* = 0, the hidden units are initialized to zeros or small random values. Then, at a time step where *t* > 0, the hidden units receive their input from the data point at the current time, $\boldsymbol{x}^{(t)}$, and the previous values of hidden units at *t* – 1, indicated as $\boldsymbol{h}^{(t-1)}$.

Similarly, in the case of a multilayer RNN, we can summarize the information flow as follows:

- *layer* = 1: Here, the hidden layer is represented as $\boldsymbol{h}_1^{(t)}$ and it receives its input from the data point, $\boldsymbol{x}^{(t)}$, and the hidden values in the same layer, but at the previous time step, $\boldsymbol{h}_1^{(t-1)}$.
- *layer* = 2: The second hidden layer, $\boldsymbol{h}_2^{(t)}$, receives its inputs from the outputs of the layer below at the current time step ($\boldsymbol{o}_1^{(t)}$) and its own hidden values from the previous time step, $\boldsymbol{h}_2^{(t-1)}$.

Since, in this case, each recurrent layer must receive a sequence as input, all the recurrent layers except the last one must *return a sequence as output* (that is, `return_sequences=True`). The behavior of the last recurrent layer depends on the type of problem.

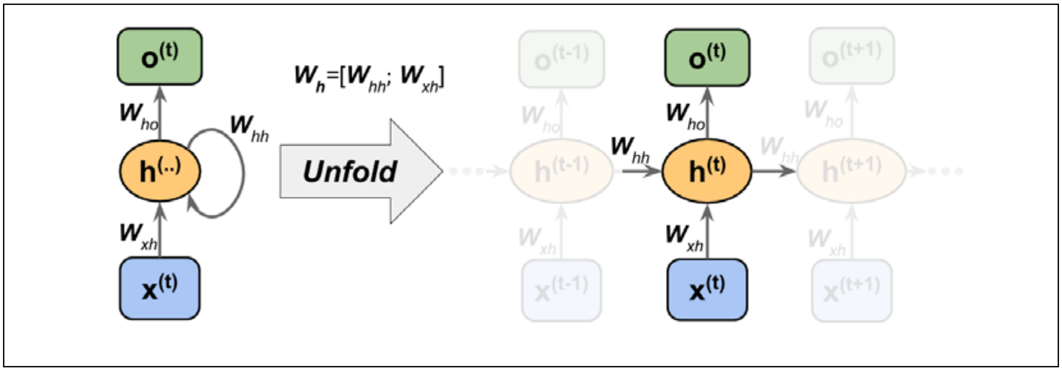
Computing activations in an RNN

Now that you understand the structure and general flow of information in an RNN, let's get more specific and compute the actual activations of the hidden layers, as well as the output layer. For simplicity, we will consider just a single hidden layer; however, the same concept applies to multilayer RNNs.

Each directed edge (the connections between boxes) in the representation of an RNN that we just looked at is associated with a weight matrix. Those weights do not depend on time, *t*; therefore, they are shared across the time axis. The different weight matrices in a single-layer RNN are as follows:

- $\boldsymbol{W}_{xh}$: The weight matrix between the input, $\boldsymbol{x}^{(t)}$, and the hidden layer, $\boldsymbol{h}$
- $\boldsymbol{W}_{hh}$: The weight matrix associated with the recurrent edge
- $\boldsymbol{W}_{ho}$: The weight matrix between the hidden layer and output layer

These weight matrices are depicted in the following figure:



In certain implementations, you may observe that the weight matrices, $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hh}$, are concatenated to a combined matrix, $\boldsymbol{W}_h = [\boldsymbol{W}_{xh}\,; \boldsymbol{W}_{hh}]$. Later in this section, we will make use of this notation as well.

Computing the activations is very similar to standard multilayer perceptrons and other types of feedforward NNs. For the hidden layer, the net input, $\boldsymbol{z}_h$ (preactivation), is computed through a linear combination, that is, we compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit:

$$\boldsymbol{z}_h^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h$$

Then, the activations of the hidden units at the time step, t, are calculated as follows:

$$\boldsymbol{h}^{(t)} = \phi_h\left(\boldsymbol{z}_h^{(t)}\right) = \phi_h\left(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right)$$

Here, $\boldsymbol{b}_h$ is the bias vector for the hidden units and $\phi_h(\cdot)$ is the activation function of the hidden layer.

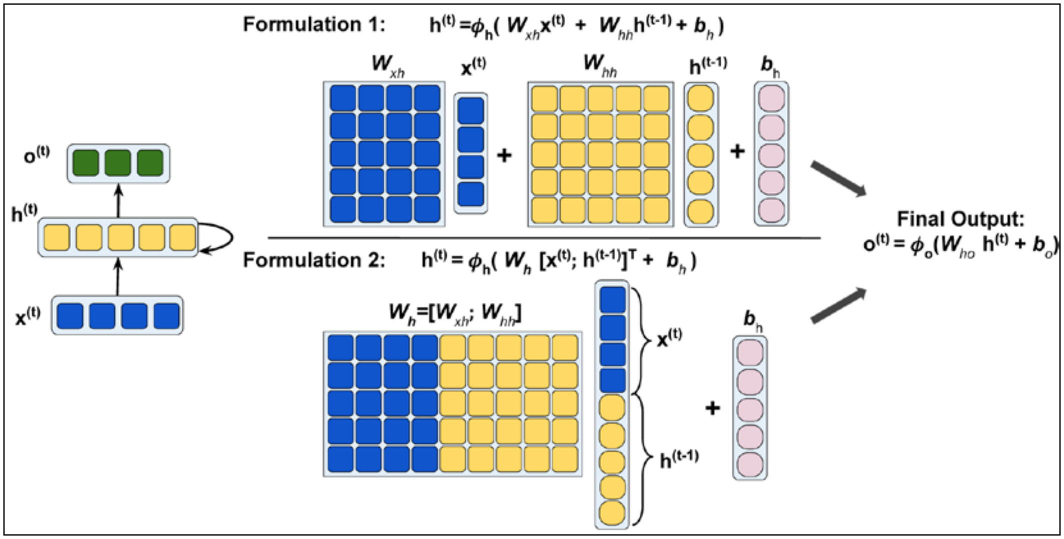
In case you want to use the concatenated weight matrix, $\boldsymbol{W}_h = [\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}]$, the formula for computing hidden units will change, as follows:

$$\boldsymbol{h}^{(t)} = \phi_h\left([\boldsymbol{W}_{xh}; \boldsymbol{W}_{hh}]\begin{bmatrix}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t-1)}\end{bmatrix} + \boldsymbol{b}_h\right)$$

Once the activations of the hidden units at the current time step are computed, then the activations of the output units will be computed, as follows:

$$\boldsymbol{o}^{(t)} = \phi_o\left(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t)} + \boldsymbol{b}_o\right)$$

To help clarify this further, the following figure shows the process of computing these activations with both formulations:


Formulation 1: $h^{(t)} = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$
W_{xh}
$x^{(t)}$
W_{hh}
$h^{(t-1)}$
b_h
+
+
$o^{(t)}$
$h^{(t)}$
$x^{(t)}$
Formulation 2: $h^{(t)} = \phi_h(W_h [x^{(t)}; h^{(t-1)}]^T + b_h)$
$W_h=[W_{xh}; W_{hh}]$
$x^{(t)}$
$h^{(t-1)}$
b_h
+
Final Output:
$o^{(t)} = \phi_o(W_{ho}\ h^{(t)} + b_o)$

Training RNNs using backpropogation through time (BPTT)

The learning algorithm for RNNs was introduced in 1990: *Backpropagation Through Time: What It Does and How to Do It* (*Paul Werbos*, *Proceedings of IEEE*, 78(10): 1550-1560, *1990*).

The derivation of the gradients might be a bit complicated, but the basic idea is that the overall loss, *L*, is the sum of all the loss functions at times *t* = 1 to *t* = *T*:

$$L = \sum_{t=1}^{T} L^{(t)}$$

Since the loss at time *t* is dependent on the hidden units at all previous time steps 1 : *t*, the gradient will be computed as follows:

$$\frac{\partial\, L^{(t)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \times \left(\sum_{k=1}^{t} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \times \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}_{hh}} \right)$$

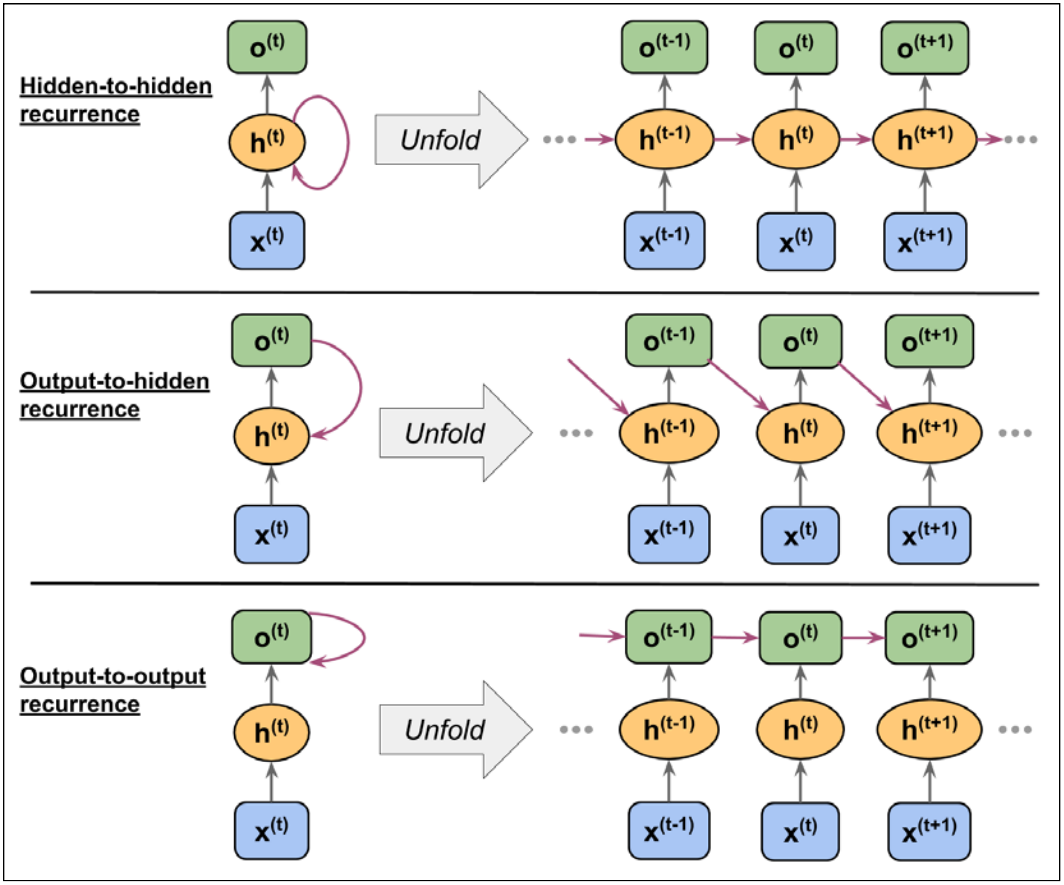
Here, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ is computed as a multiplication of adjacent time steps:

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k+1}^{t} \frac{\partial\, \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

Hidden-recurrence versus output-recurrence

So far, you have seen recurrent networks in which the hidden layer has the recurrent property. However, note that there is an alternative model in which the recurrent connection comes from the output layer. In this case, the net activations from the output layer at the previous time step, $\boldsymbol{o}^{t-1}$, can be added in one of two ways:

- To the hidden layer at the current time step, $\boldsymbol{h}^{t}$ (shown in the following figure as output-to-hidden recurrence)
- To the output layer at the current time step, $\boldsymbol{o}^{t}$ (shown in the following figure as output-to-output recurrence)



As shown in the previous figure, the differences between these architectures can be clearly seen in the recurring connections. Following our notation, the weights associated with the recurrent connection will be denoted for the hidden-to-hidden recurrence by $\boldsymbol{W}_{hh}$, for the output-to-hidden recurrence by $\boldsymbol{W}_{oh}$, and for the output-to-output recurrence by $\boldsymbol{W}_{oo}$. In some articles in literature, the weights associated with the recurrent connections are also denoted by $\boldsymbol{W}_{rec}$.

To see how this works in practice, let's manually compute the forward pass for one of these recurrent types. Using the TensorFlow Keras API, a recurrent layer can be defined via `SimpleRNN`, which is similar to the output-to-output recurrence. In the following code, we will create a recurrent layer from `SimpleRNN` and perform a forward pass on an input sequence of length 3 to compute the output. We will also manually compute the forward pass and compare the results with those of `SimpleRNN`. First, let's create the layer and assign the weights for our manual computations:

```
>>> import tensorflow as tf
>>> tf.random.set_seed(1)

>>> rnn_layer = tf.keras.layers.SimpleRNN(
...     units=2, use_bias=True,
...     return_sequences=True)
>>> rnn_layer.build(input_shape=(None, None, 5))

>>> w_xh, w_oo, b_h = rnn_layer.weights

>>> print('W_xh shape:', w_xh.shape)
>>> print('W_oo shape:', w_oo.shape)
>>> print('b_h  shape:', b_h.shape)
W_xh shape: (5, 2)
W_oo shape: (2, 2)
b_h  shape: (2,)
```

The input shape for this layer is `(None, None, 5)`, where the first dimension is the batch dimension (using `None` for variable batch size), the second dimension corresponds to the sequence (using `None` for the variable sequence length), and the last dimension corresponds to the features. Notice that we set `return_sequences=True`, which, for an input sequence of length 3, will result in the output sequence $\langle \boldsymbol{o}^{(0)}, \boldsymbol{o}^{(1)}, \boldsymbol{o}^{(2)} \rangle$. Otherwise, it would only return the final output, $\boldsymbol{o}^{(2)}$.

Now, we will call the forward pass on the `rnn_layer` and manually compute the outputs at each time step and compare them:

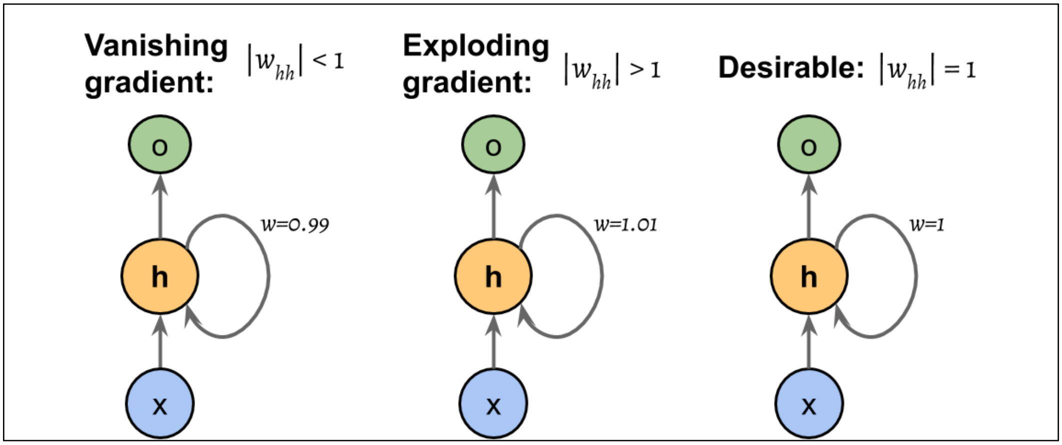
```
>>> x_seq = tf.convert_to_tensor(
...     [[1.0]*5, [2.0]*5, [3.0]*5],
...     dtype=tf.float32)

>>> ## output of SimepleRNN:
>>> output = rnn_layer(tf.reshape(x_seq, shape=(1, 3, 5)))

>>> ## manually computing the output:
>>> out_man = []
>>> for t in range(len(x_seq)):
...     xt = tf.reshape(x_seq[t], (1, 5))
...     print('Time step {} =>'.format(t))
...     print('   Input          :', xt.numpy())
...
...     ht = tf.matmul(xt, w_xh) + b_h
...     print('   Hidden         :', ht.numpy())
...
...     if t>0:
```

```
...         prev_o = out_man[t-1]
...     else:
...         prev_o = tf.zeros(shape=(ht.shape))
...     ot = ht + tf.matmul(prev_o, w_oo)
...     ot = tf.math.tanh(ot)
...     out_man.append(ot)
...     print('   Output (manual) :', ot.numpy())
...     print('   SimpleRNN output:'.format(t),
...           output[0][t].numpy())
...     print()
Time step 0 =>
   Input           : [[1. 1. 1. 1. 1.]]
   Hidden          : [[0.41464037 0.96012145]]
   Output (manual) : [[0.39240566 0.74433106]]
   SimpleRNN output: [0.39240566 0.74433106]

Time step 1 =>
   Input           : [[2. 2. 2. 2. 2.]]
   Hidden          : [[0.82928073 1.9202429 ]]
   Output (manual) : [[0.80116504 0.9912947 ]]
   SimpleRNN output: [0.80116504 0.9912947 ]

Time step 2 =>
   Input           : [[3. 3. 3. 3. 3.]]
   Hidden          : [[1.243921  2.8803642]]
   Output (manual) : [[0.95468265 0.9993069 ]]
   SimpleRNN output: [0.95468265 0.9993069 ]
```

In our manual forward computation, we used the hyperbolic tangent (tanh) activation function, since it is also used in `SimpleRNN` (the default activation). As you can see from the printed results, the outputs from the manual forward computations exactly match the output of the `SimpleRNN` layer at each time step. Hopefully, this hands-on task has enlightened you on the mysteries of recurrent networks.

The challenges of learning long-range interactions

BPTT, which was briefly mentioned earlier, introduces some new challenges. Because of the multiplicative factor, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$, in computing the gradients of a loss function, the so-called **vanishing** and **exploding** gradient problems arise. These problems are explained by the examples in the following figure, which shows an RNN with only one hidden unit for simplicity:


Vanishing gradient: $|w_{hh}| < 1$
Exploding gradient: $|w_{hh}| > 1$
Desirable: $|w_{hh}| = 1$
o
h
x
w=0.99
o
h
x
w=1.01
o
h
x
w=1

Basically, $\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}$ has $t - k$ multiplications; therefore, multiplying the weight, w, by itself $t - k$ times results in a factor, w^{t-k}. As a result, if $|w| < 1$, this factor becomes very small when $t - k$ is large. On the other hand, if the weight of the recurrent edge is $|w| > 1$, then w^{t-k} becomes very large when $t - k$ is large. Note that large $t - k$ refers to long-range dependencies. We can see that a naive solution to avoid vanishing or exploding gradients can be reached by ensuring $|w| = 1$. If you are interested and would like to investigate this in more detail, read *On the difficulty of training recurrent neural networks*, by *R. Pascanu*, *T. Mikolov*, and *Y. Bengio*, *2012* (`https://arxiv.org/pdf/1211.5063.pdf`).

In practice, there are at least three solutions to this problem:

- Gradient clipping
- TBPTT
- LSTM

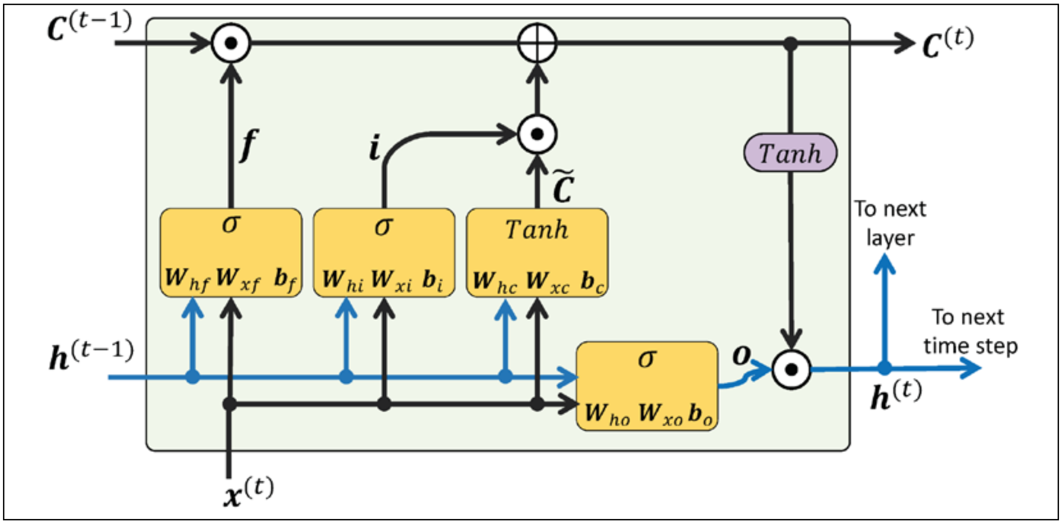
Using gradient clipping, we specify a cut-off or threshold value for the gradients, and we assign this cut-off value to gradient values that exceed this value. In contrast, TBPTT simply limits the number of time steps that the signal can backpropagate after each forward pass. For example, even if the sequence has 100 elements or steps, we may only backpropagate the most recent 20 time steps.

While both gradient clipping and TBPTT can solve the exploding gradient problem, the truncation limits the number of steps that the gradient can effectively flow back and properly update the weights. On the other hand, LSTM, designed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber, has been more successful in vanishing and exploding gradient problems while modeling long-range dependencies through the use of memory cells. Let's discuss LSTM in more detail.

Long short-term memory cells

As stated previously, LSTMs were first introduced to overcome the vanishing gradient problem (*Long Short-Term Memory*, *S. Hochreiter* and *J. Schmidhuber*, *Neural Computation*, 9(8): 1735-1780, *1997*). The building block of an LSTM is a **memory cell**, which essentially represents or replaces the hidden layer of standard RNNs.

In each memory cell, there is a recurrent edge that has the desirable weight, $w = 1$, as we discussed, to overcome the vanishing and exploding gradient problems. The values associated with this recurrent edge are collectively called the **cell state**. The unfolded structure of a modern LSTM cell is shown in the following figure:



Notice that the cell state from the previous time step, $\boldsymbol{C}^{(t-1)}$, is modified to get the cell state at the current time step, $\boldsymbol{C}^{(t)}$, without being multiplied directly with any weight factor. The flow of information in this memory cell is controlled by several computation units (often called *gates*) that will be described here. In the previous figure, $\odot$ refers to the **element-wise product** (element-wise multiplication) and $\oplus$ means **element-wise summation** (element-wise addition). Furthermore, $\boldsymbol{x}^{(t)}$ refers to the input data at time *t*, and $\boldsymbol{h}^{(t-1)}$ indicates the hidden units at time *t* – 1. Four boxes are indicated with an activation function, either the sigmoid function (σ) or tanh, and a set of weights; these boxes apply a linear combination by performing matrix-vector multiplications on their inputs (which are $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{x}^{(t)}$). These units of computation with sigmoid activation functions, whose output units are passed through $\odot$, are called gates.

In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate, and the output gate:

- The **forget gate** ($\boldsymbol{f}_t$) allows the memory cell to reset the cell state without growing indefinitely. In fact, the forget gate decides which information is allowed to go through and which information to suppress. Now, $\boldsymbol{f}_t$ is computed as follows:

  $$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right)$$

  Note that the forget gate was not part of the original LSTM cell; it was added a few years later to improve the original model (*Learning to Forget: Continual Prediction with LSTM*, *F. Gers*, *J. Schmidhuber*, and *F. Cummins*, *Neural Computation 12*, 2451-2471, *2000*).

- The **input gate** ($\boldsymbol{i}_t$) and **candidate value** ($\widetilde{\boldsymbol{C}}_t$) are responsible for updating the cell state. They are computed as follows:

  $$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \ \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right)$$

  $$\widetilde{\boldsymbol{C}}_t = \tanh\left(\boldsymbol{W}_{xc}\,\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hc}\,\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c\right)$$

  The cell state at time *t* is computed as follows:

  $$\boldsymbol{C}^{(t)} = \left(\boldsymbol{C}^{(t-1)} \odot \boldsymbol{f}_t\right) \oplus \left(\boldsymbol{i}_t \odot \widetilde{\boldsymbol{C}}_t\right)$$

- The **output gate** ($\boldsymbol{o}_t$) decides how to update the values of hidden units:

  $$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\,\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right)$$

Given this, the hidden units at the current time step are computed as follows:

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right)$$

The structure of an LSTM cell and its underlying computations might seem very complex and hard to implement. However, the good news is that TensorFlow has already implemented everything in optimized wrapper functions, which allows us to define our LSTM cells easily and efficiently. We will apply RNNs and LSTMs to real-world datasets later in this chapter.

Other advanced RNN models

LSTMs provide a basic approach for modeling long-range dependencies in sequences. Yet, it is important to note that there are many variations of LSTMs described in literature (*An Empirical Exploration of Recurrent Network Architectures*, *Rafal Jozefowicz*, *Wojciech Zaremba*, and *Ilya Sutskever*, *Proceedings of ICML*, 2342-2350, *2015*). Also worth noting is a more recent approach, Gated Recurrent Unit (GRU), which was proposed in 2014. GRUs have a simpler architecture than LSTMs; therefore, they are computationally more efficient, while their performance in some tasks, such as polyphonic music modeling, is comparable to LSTMs. If you are interested in learning more about these modern RNN architectures, refer to the paper, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*, by *Junyoung Chung* and others, *2014* (`https://arxiv.org/pdf/1412.3555v1.pdf`).

Implementing RNNs for sequence modeling in TensorFlow

Now that we have covered the underlying theory behind RNNs, we are ready to move on to the more practical portion of this chapter: implementing RNNs in TensorFlow. During the rest of this chapter, we will apply RNNs to two common problem tasks:

1. Sentiment analysis
2. Language modeling

These two projects, which we will walk through together in the following pages, are both fascinating but also quite involved. Thus, instead of providing the code all at once, we will break the implementation up into several steps and discuss the code in detail. If you like to have a big picture overview and want to see all the code at once before diving into the discussion, take a look at the code implementation first, which you can view at `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch16`.

Project one – predicting the sentiment of IMDb movie reviews

You may recall from *Chapter 8, Applying Machine Learning to Sentiment Analysis*, that sentiment analysis is concerned with analyzing the expressed opinion of a sentence or a text document. In this section and the following subsections, we will implement a multilayer RNN for sentiment analysis using a many-to-one architecture.

In the next section, we will implement a many-to-many RNN for an application of language modeling. While the chosen examples are purposefully simple to introduce the main concepts of RNNs, language modeling has a wide range of interesting applications, such as building chatbots—giving computers the ability to directly talk and interact with humans.

Preparing the movie review data

In the preprocessing steps in *Chapter 8*, we created a clean dataset named `movie_data.csv`, which we will use again now. First, we will import the necessary modules and read the data into a pandas `DataFrame`, as follows:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> import numpy as np
>>> import pandas as pd

>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Remember that this data frame, `df`, consists of two columns, namely `'review'` and `'sentiment'`, where `'review'` contains the text of movie reviews (the input features), and `'sentiment'` represents the target label we want to predict (`0` refers to negative sentiment and `1` refers to positive sentiment). The text component of these movie reviews is sequences of words, and the RNN model classifies each sequence as a positive (`1`) or negative (`0`) review.

However, before we can feed the data into an RNN model, we need to apply several preprocessing steps:

1. Create a TensorFlow dataset object and split it into separate training, testing, and validation partitions.
2. Identify the unique words in the training dataset.

3. Map each unique word to a unique integer and encode the review text into encoded integers (an index of each unique word).
4. Divide the dataset into mini-batches as input to the model.

Let's proceed with the first step: creating a TensorFlow dataset from this data frame:

```
>>> ## Step 1: create a dataset
>>> target = df.pop('sentiment')
>>> ds_raw = tf.data.Dataset.from_tensor_slices(
...     (df.values, target.values))

>>> ## inspection:
>>> for ex in ds_raw.take(3):
...     tf.print(ex[0].numpy()[0][ :50], ex[1])

b'In 1974, the teenager Martha Moxley (Maggie Grace)' 1
b'OK... so... I really like Kris Kristofferson and h' 0
b'***SPOILER*** Do not read this, if you think about' 0
```

Now, we can split it into training, testing, and validation datasets. The entire dataset contains 50,000 examples. We will keep the first 25,000 examples for evaluation (hold-out testing dataset), and then 20,000 examples will be used for training and 5,000 for validation. The code is as follows:

```
>>> tf.random.set_seed(1)
>>> ds_raw = ds_raw.shuffle(
...     50000, reshuffle_each_iteration=False)

>>> ds_raw_test = ds_raw.take(25000)
>>> ds_raw_train_valid = ds_raw.skip(25000)
>>> ds_raw_train = ds_raw_train_valid.take(20000)
>>> ds_raw_valid = ds_raw_train_valid.skip(20000)
```

To prepare the data for input to a NN, we need to encode it into numeric values, as was mentioned in steps 2 and 3. To do this, we will first find the unique words (tokens) in the training dataset. While finding unique tokens is a process for which we can use Python datasets, it can be more efficient to use the `Counter` class from the `collections` package, which is part of Python's standard library.

In the following code, we will instantiate a new `Counter` object (`token_counts`) that will collect the unique word frequencies. Note that in this particular application (and in contrast to the bag-of-words model), we are only interested in the set of unique words and won't require the word counts, which are created as a side product. To split the text into words (or tokens), the `tensorflow_datasets` package provides a `Tokenizer` class.

The code for collecting unique tokens is as follows:

```
>>> ## Step 2: find unique tokens (words)
>>> from collections import Counter

>>> tokenizer = tfds.features.text.Tokenizer()
>>> token_counts = Counter()

>>> for example in ds_raw_train:
...     tokens = tokenizer.tokenize(example[0].numpy()[0])
...     token_counts.update(tokens)

>>> print('Vocab-size:', len(token_counts))
Vocab-size: 87007
```

If you want to learn more about `Counter`, refer to its documentation at `https://docs.python.org/3/library/collections.html#collections.Counter`.

Next, we are going to map each unique word to a unique integer. This can be done manually using a Python dictionary, where the keys are the unique tokens (words) and the value associated with each key is a unique integer. However, the `tensorflow_datasets` package already provides a class, `TokenTextEncoder`, which we can use to create such a mapping and encode the entire dataset. First, we will create an `encoder` object from the `TokenTextEncoder` class by passing the unique tokens (`token_counts` contains the tokens and their counts, although here, their counts are not needed, so they will be ignored). Calling the `encoder.encode()` method will then convert its input text into a list of integer values:

```
>>> ## Step 3: encoding unique tokens to integers
>>> encoder = tfds.features.text.TokenTextEncoder(token_counts)
>>> example_str = 'This is an example!'
>>> print(encoder.encode(example_str))
[232, 9, 270, 1123]
```

Note that there might be some tokens in the validation or testing data that are not present in the training data and are thus not included in the mapping. If we have q tokens (that is the size of `token_counts` passed to the `TokenTextEncoder`, which in this case is 87,007), then all tokens that haven't been seen before, and are thus not included in `token_counts`, will be assigned the integer $q + 1$ (which will be 87,008 in our case). In other words, the index $q + 1$ is reserved for unknown words. Another reserved value is the integer 0, which serves as a placeholder for adjusting the sequence length. Later, when we are building an RNN model in TensorFlow, we will consider these two placeholders, 0 and $q + 1$, in more detail.

We can use the `map()` method of the dataset objects to transform each text in the dataset accordingly, just like we would apply any other transformation to a dataset. However, there is a small problem: here, the text data is enclosed in tensor objects, which we can access by calling the `numpy()` method on a tensor in the eager execution mode. But during transformations by the `map()` method, the eager execution will be disabled. To solve this problem, we can define two functions. The first function will treat the input tensors as if the eager execution mode is enabled:

```
>>> ## Step 3-A: define the function for transformation
>>> def encode(text_tensor, label):
...     text = text_tensor.numpy()[0]
...     encoded_text = encoder.encode(text)
...     return encoded_text, label
```

In the second function, we will wrap the first function using `tf.py_function` to convert it into a TensorFlow operator, which can then be used via its `map()` method. This process of encoding text into a list of integers can be carried out using the following code:

```
>>> ## Step 3-B: wrap the encode function to a TF Op.
>>> def encode_map_fn(text, label):
...     return tf.py_function(encode, inp=[text, label],
...                           Tout=(tf.int64, tf.int64))

>>> ds_train = ds_raw_train.map(encode_map_fn)
>>> ds_valid = ds_raw_valid.map(encode_map_fn)
>>> ds_test = ds_raw_test.map(encode_map_fn)

>>> # look at the shape of some examples:
>>> tf.random.set_seed(1)
>>> for example in ds_train.shuffle(1000).take(5):
...     print('Sequence length:', example[0].shape)
Sequence length: (24,)
Sequence length: (179,)
Sequence length: (262,)
Sequence length: (535,)
Sequence length: (130,)
```

So far, we've converted sequences of words into sequences of integers. However, there is one issue that we still need to resolve—the sequences currently have different lengths (as shown in the result of executing the previous code for five randomly chosen examples). Although, in general, RNNs can handle sequences with different lengths, we still need to make sure that all the sequences in a mini-batch have the same length to store them efficiently in a tensor.

To divide a dataset that has elements with different shapes into mini-batches, TensorFlow provides a different method, `padded_batch()` (instead of `batch()`), which will automatically pad the consecutive elements that are to be combined into a batch with placeholder values (0s) so that all sequences within a batch will have the same shape. To illustrate this with a practical example, let's take a small subset of size 8 from the training dataset, `ds_train`, and apply the `padded_batch()` method to this subset with `batch_size=4`. We will also print the sizes of the individual elements before combining these into mini-batches, as well as the dimensions of the resulting mini-batches:

```
>>> ## Take a small subset
>>> ds_subset = ds_train.take(8)
>>> for example in ds_subset:
...     print('Individual size:', example[0].shape)
Individual size: (119,)
Individual size: (688,)
Individual size: (308,)
Individual size: (204,)
Individual size: (326,)
Individual size: (240,)
Individual size: (127,)
Individual size: (453,)

>>> ## Dividing the dataset into batches
>>> ds_batched = ds_subset.padded_batch(
...              4, padded_shapes=([-1], []))

>>> for batch in ds_batched:
...     print('Batch dimension:', batch[0].shape)
Batch dimension: (4, 688)
Batch dimension: (4, 453)
```

As you can observe from the printed tensor shapes, the number of columns (that is, `.shape[1]`) in the first batch is 688, which resulted from combining the first four examples into a single batch and using the maximum size of these examples. That means that the other three examples in this batch are padded as much as necessary to match this size. Similarly, the second batch keeps the maximum size of its individual four examples, which is 453, and pads the other examples so that their length is smaller than the maximum length.

Let's divide all three datasets into mini-batches with a batch size of 32:

```
>>> train_data = ds_train.padded_batch(
...     32, padded_shapes=([-1],[]))
```

```
>>> valid_data = ds_valid.padded_batch(
...     32, padded_shapes=([-1],[]))

>>> test_data = ds_test.padded_batch(
...     32, padded_shapes=([-1],[]))
```

Now, the data is in a suitable format for an RNN model, which we are going to implement in the following subsections. In the next subsection, however, we will first discuss feature **embedding**, which is an optional but highly recommended preprocessing step that is used for reducing the dimensionality of the word vectors.

Embedding layers for sentence encoding

During the data preparation in the previous step, we generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the *indices* of unique words. These word indices can be converted into input features in several different ways. One naive way is to apply one-hot encoding to convert the indices into vectors of zeros and ones. Then, each word will be mapped to a vector whose size is the number of unique words in the entire dataset. Given that the number of unique words (the size of the vocabulary) can be in the order of $10^4 - 10^5$, which will also be the number of our input features, a model trained on such features may suffer from the **curse of dimensionality**. Furthermore, these features are very sparse, since all are zero except one.

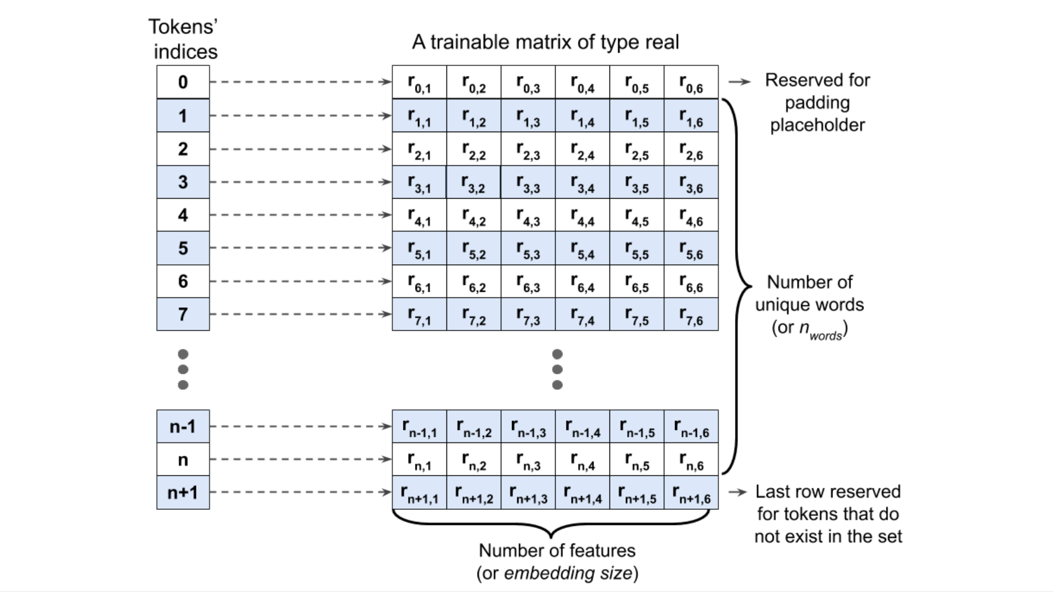
A more elegant approach is to map each word to a vector of a fixed size with real-valued elements (not necessarily integers). In contrast to the one-hot encoded vectors, we can use finite-sized vectors to represent an infinite number of real numbers. (In theory, we can extract infinite real numbers from a given interval, for example [–1, 1].)

This is the idea behind embedding, which is a feature-learning technique that we can utilize here to automatically learn the salient features to represent the words in our dataset. Given the number of unique words, n_{words}, we can select the size of the embedding vectors (a.k.a., embedding dimension) to be much smaller than the number of unique words ($embedding\_dim << n_{words}$) to represent the entire vocabulary as input features.

The advantages of embedding over one-hot encoding are as follows:

- A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality
- The extraction of salient features since the embedding layer in an NN can be optimized (or learned)

The following schematic representation shows how embedding works by mapping token indices to a trainable embedding matrix:


Tokens' indices
A trainable matrix of type real
0
1
2
3
4
5
6
7
n-1
n
n+1
$r_{0,1}$ $r_{0,2}$ $r_{0,3}$ $r_{0,4}$ $r_{0,5}$ $r_{0,6}$
$r_{1,1}$ $r_{1,2}$ $r_{1,3}$ $r_{1,4}$ $r_{1,5}$ $r_{1,6}$
$r_{2,1}$ $r_{2,2}$ $r_{2,3}$ $r_{2,4}$ $r_{2,5}$ $r_{2,6}$
$r_{3,1}$ $r_{3,2}$ $r_{3,3}$ $r_{3,4}$ $r_{3,5}$ $r_{3,6}$
$r_{4,1}$ $r_{4,2}$ $r_{4,3}$ $r_{4,4}$ $r_{4,5}$ $r_{4,6}$
$r_{5,1}$ $r_{5,2}$ $r_{5,3}$ $r_{5,4}$ $r_{5,5}$ $r_{5,6}$
$r_{6,1}$ $r_{6,2}$ $r_{6,3}$ $r_{6,4}$ $r_{6,5}$ $r_{6,6}$
$r_{7,1}$ $r_{7,2}$ $r_{7,3}$ $r_{7,4}$ $r_{7,5}$ $r_{7,6}$
$r_{n-1,1}$ $r_{n-1,2}$ $r_{n-1,3}$ $r_{n-1,4}$ $r_{n-1,5}$ $r_{n-1,6}$
$r_{n,1}$ $r_{n,2}$ $r_{n,3}$ $r_{n,4}$ $r_{n,5}$ $r_{n,6}$
$r_{n+1,1}$ $r_{n+1,2}$ $r_{n+1,3}$ $r_{n+1,4}$ $r_{n+1,5}$ $r_{n+1,6}$
Reserved for padding placeholder
Number of unique words (or n_{words})
Last row reserved for tokens that do not exist in the set
Number of features (or *embedding size*)

Given a set of tokens of size *n* + 2 (*n* is the size of the token set, plus index 0 is reserved for the padding placeholder, and *n* + 1 is for the words not present in the token set), an embedding matrix of size $(n + 2) \times embedding\_dim$ will be created where each row of this matrix represents numeric features associated with a token. Therefore, when an integer index, *i*, is given as input to the embedding, it will look up the corresponding row of the matrix at index *i* and return the numeric features. The embedding matrix serves as the input layer to our NN models.

In practice, creating an embedding layer can simply be done using `tf.keras.layers.Embedding`. Let's see an example where we will create a model and add an embedding layer, as follows:

```
>>> from tensorflow.keras.layers import Embedding

>>> model = tf.keras.Sequential()

>>> model.add(Embedding(input_dim=100,
...                     output_dim=6,
...                     input_length=20,
...                     name='embed-layer'))

>>> model.summary()
Model: "sequential"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embed-layer (Embedding)      (None, 20, 6)             600
=================================================================
Total params: 6,00
Trainable params: 6,00
Non-trainable params: 0
_________________________________________________________________
```

The input to this model (embedding layer) must have rank 2 with dimensionality $batchsize \times input\_length$, where $input\_length$ is the length of sequences (here, set to 20 via the `input_length` argument). For example, an input sequence in the mini-batch could be $\langle 14, 43, 52, 61, 8,19,67,83,10,7,42,87,56,18,94,17,67,90,6,39 \rangle$, where each element of this sequence is the index of the unique words. The output will have dimensionality $batchsize \times input\_length \times embedding\_dim$, where $embedding\_dim$ is the size of the embedding features (here, set to 6 via `output_dim`). The other argument provided to the embedding layer, `input_dim`, corresponds to the unique integer values that the model will receive as input (for instance, $n + 2$, set here to 100). Therefore, the embedding matrix in this case has the size 100×6.

Dealing with variable sequence lengths
Note that the `input_length` argument is not required, and we can use `None` for cases where the lengths of input sequences vary. You can find more information about this function in the official documentation at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding`.

Building an RNN model

Now we're ready to build an RNN model. Using the Keras `Sequential` class, we can combine the embedding layer, the recurrent layers of the RNN, and the fully connected non-recurrent layers. For the recurrent layers, we can use any of the following implementations:

- `SimpleRNN`: a regular RNN layer, that is, a fully connected recurrent layer
- `LSTM`: a long short-term memory RNN, which is useful for capturing the long-term dependencies
- `GRU`: a recurrent layer with a gated recurrent unit, as proposed in *Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation* (`https://arxiv.org/abs/1406.1078v3`), as an alternative to LSTMs

To see how a multilayer RNN model can be built using one of these recurrent layers, in the following example, we will create an RNN model, starting with an embedding layer with `input_dim=1000` and `output_dim=32`. Then, two recurrent layers of type `SimpleRNN` will be added. Finally, we will add a non-recurrent fully connected layer as the output layer, which will return a single output value as the prediction:

```
>>> from tensorflow.keras import Sequential
>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import Dense

>>> model = Sequential()
>>> model.add(Embedding(input_dim=1000, output_dim=32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32))
>>> model.add(Dense(1))
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 32)          32000
_________________________________________________________________
simple_rnn (SimpleRNN)       (None, None, 32)          2080
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 36,193
Trainable params: 36,193
Non-trainable params: 0
_________________________________________________________________
```

As you can see, building an RNN model using these recurrent layers is pretty straightforward. In the next subsection, we will go back to our sentiment analysis task and build an RNN model to solve that.

Building an RNN model for the sentiment analysis task

Since we have very long sequences, we are going to use an LSTM layer to account for long-term effects. In addition, we will put the LSTM layer inside a `Bidirectional` wrapper, which will make the recurrent layers pass through the input sequences from both directions, start to end, as well as the reverse direction:

```
>>> embedding_dim = 20
>>> vocab_size = len(token_counts) + 2

>>> tf.random.set_seed(1)

>>> ## build the model
>>> bi_lstm_model = tf.keras.Sequential([
...     tf.keras.layers.Embedding(
...         input_dim=vocab_size,
...         output_dim=embedding_dim,
...         name='embed-layer'),
...
...     tf.keras.layers.Bidirectional(
...         tf.keras.layers.LSTM(64, name='lstm-layer'),
...         name='bidir-lstm'),
...
...     tf.keras.layers.Dense(64, activation='relu'),
...
...     tf.keras.layers.Dense(1, activation='sigmoid')
>>> ])

>>> bi_lstm_model.summary()

>>> ## compile and train:
>>> bi_lstm_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
...     metrics=['accuracy'])

>>> history = bi_lstm_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)
```

```
>>> ## evaluate on the test data
>>> test_results = bi_lstm_model.evaluate(test_data)
>>> print('Test Acc.: {:.2f}%'.format(test_results[1]*100))
Epoch 1/10
625/625 [==============================] - 96s 154ms/step - loss: 
0.4410 - accuracy: 0.7782 - val_loss: 0.0000e+00 - val_accuracy: 
0.0000e+00
Epoch 2/10
625/625 [==============================] - 95s 152ms/step - loss: 
0.1799 - accuracy: 0.9326 - val_loss: 0.4833 - val_accuracy: 0.8414
. . .

Test Acc.: 85.15%
```

After training this model for 10 epochs, evaluation on the test data shows 85 percent accuracy. (Note that this result is not the best when compared to the state-of-the-art methods used on the IMDb dataset. The goal was simply to show how RNN works.)

More on the bidirectional RNN

The `Bidirectional` wrapper makes two passes over each input sequence: a forward pass and a reverse or backward pass (note that this is not to be confused with the forward and backward passes in the context of backpropagation). The results of these forward and backward passes will be concatenated by default. But if you want to change this behavior, you can set the argument `merge_mode` to `'sum'` (for summation), `'mul'` (for multiplying the results of the two passes), `'ave'` (for taking the average of the two), `'concat'` (which is the default), or `None`, which returns the two tensors in a list. For more information about the `Bidirectional` wrapper, feel free to look at the official documentation at `https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Bidirectional`.

We can also try other types of recurrent layers, such as `SimpleRNN`. However, as it turns out, a model built with regular recurrent layers won't be able to reach a good predictive performance (even on the training data). For example, if you try replacing the bidirectional LSTM layer in the previous code with a unidirectional `SimpleRNN` layer and train the model on full-length sequences, you may observe that the loss will not even decrease during training. The reason is that the sequences in this dataset are too long, so a model with a `SimpleRNN` layer cannot learn the long-term dependencies and may suffer from vanishing or exploding gradient problems.

In order to obtain reasonable predictive performance on this dataset using a SimpleRNN, we can truncate the sequences. Also, utilizing our "domain knowledge," we may hypothesize that the last paragraphs of a movie review may contain most of the information about its sentiment. Hence, we can focus only on the last portion of each review. To do this, we will define a helper function, preprocess_datasets(), to combine the preprocessing steps 2-4. An optional argument to this function is max_seq_length, which determines how many tokens from each review should be used. For example, if we set max_seq_length=100 and a review has more than 100 tokens, only the last 100 tokens will be used. If max_seq_length is set to None, then full-length sequences will be used as before. Trying different values for max_seq_length will give us more insights on the capability of different RNN models to handle long sequences.

The code for the preprocess_datasets() function is as follows:

```
>>> from collections import Counter

>>> def preprocess_datasets(
...     ds_raw_train,
...     ds_raw_valid,
...     ds_raw_test,
...     max_seq_length=None,
...     batch_size=32):
...
...     ## (step 1 is already done)
...     ## Step 2: find unique tokens
...     tokenizer = tfds.features.text.Tokenizer()
...     token_counts = Counter()
...
...     for example in ds_raw_train:
...         tokens = tokenizer.tokenize(example[0].numpy()[0])
...         if max_seq_length is not None:
...             tokens = tokens[-max_seq_length:]
...         token_counts.update(tokens)
...
...     print('Vocab-size:', len(token_counts))
...
...     ## Step 3: encoding the texts
...     encoder = tfds.features.text.TokenTextEncoder(
...                   token_counts)
...     def encode(text_tensor, label):
...         text = text_tensor.numpy()[0]
...         encoded_text = encoder.encode(text)
...         if max_seq_length is not None:
```

```
...             encoded_text = encoded_text[-max_seq_length:]
...         return encoded_text, label
...
...     def encode_map_fn(text, label):
...         return tf.py_function(encode, inp=[text, label],
...                               Tout=(tf.int64, tf.int64))
...
...     ds_train = ds_raw_train.map(encode_map_fn)
...     ds_valid = ds_raw_valid.map(encode_map_fn)
...     ds_test = ds_raw_test.map(encode_map_fn)
...
...     ## Step 4: batching the datasets
...     train_data = ds_train.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     valid_data = ds_valid.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     test_data = ds_test.padded_batch(
...         batch_size, padded_shapes=([-1],[]))
...
...     return (train_data, valid_data,
...             test_data, len(token_counts))
```

Next, we will define another helper function, `build_rnn_model()`, for building models with different architectures more conveniently:

```
>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import Bidirectional
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import LSTM
>>> from tensorflow.keras.layers import GRU

>>> def build_rnn_model(embedding_dim, vocab_size,
...                     recurrent_type='SimpleRNN',
...                     n_recurrent_units=64,
...                     n_recurrent_layers=1,
...                     bidirectional=True):
...
...     tf.random.set_seed(1)
...
...     # build the model
...     model = tf.keras.Sequential()
...
...     model.add(
...         Embedding(
...             input_dim=vocab_size,
...             output_dim=embedding_dim,
...             name='embed-layer')
...     )
...
...     for i in range(n_recurrent_layers):
...         return_sequences = (i < n_recurrent_layers-1)
...
...         if recurrent_type == 'SimpleRNN':
...             recurrent_layer = SimpleRNN(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='simprnn-layer-{}'.format(i))
...         elif recurrent_type == 'LSTM':
...             recurrent_layer = LSTM(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='lstm-layer-{}'.format(i))
...         elif recurrent_type == 'GRU':
...             recurrent_layer = GRU(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='gru-layer-{}'.format(i))
...
...         if bidirectional:
...             recurrent_layer = Bidirectional(
...                 recurrent_layer, name='bidir-' +
...                 recurrent_layer.name)
...
...         model.add(recurrent_layer)
...
...     model.add(tf.keras.layers.Dense(64, activation='relu'))
...     model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
...
...     return model
```

Now, using these two fairly general, but convenient, helper functions, we can readily compare different RNN models with different input sequence lengths. As an example, in the following code, we will try a model with a single recurrent layer of type `SimpleRNN` while truncating the sequences to a maximum length of 100 tokens:

```
>>> batch_size = 32
>>> embedding_dim = 20
>>> max_seq_length = 100

>>> train_data, valid_data, test_data, n = preprocess_datasets(
...     ds_raw_train, ds_raw_valid, ds_raw_test,
...     max_seq_length=max_seq_length,
...     batch_size=batch_size
... )

>>> vocab_size = n + 2

>>> rnn_model = build_rnn_model(
...     embedding_dim, vocab_size,
...     recurrent_type='SimpleRNN',
...     n_recurrent_units=64,
...     n_recurrent_layers=1,
...     bidirectional=True)

>>> rnn_model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embed-layer (Embedding)      (None, None, 20)          1161300
_________________________________________________________________
bidir-simprnn-layer-0 (Bidir (None, 128)               10880
_________________________________________________________________
Dense    (Dense)             (None, 64)                8256
_________________________________________________________________
dense_1  (Dense)             (None, 1)                 65
=================================================================
Total params: 1,180,501
Trainable params: 1,180,501
Non-trainable params: 0
_________________________________________________________________

>>> rnn_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(
...          from_logits=False), metrics=['accuracy'])

>>> history = rnn_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)
```

```
Epoch 1/10
625/625 [==============================] - 73s 118ms/step - loss:
0.6996 - accuracy: 0.5074 - val_loss: 0.6880 - val_accuracy: 0.5476
Epoch 2/10


>>> results = rnn_model.evaluate(test_data)
>>> print('Test Acc.: {:.2f}%'.format(results[1]*100))
Test Acc.: 80.70%
```

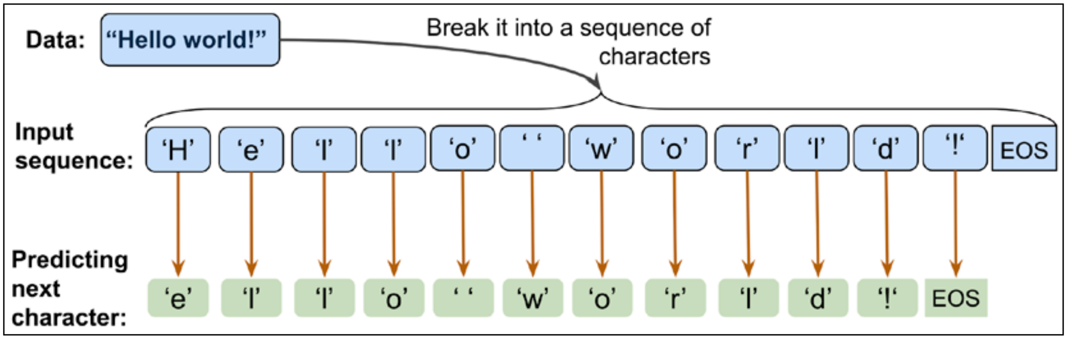
For instance, truncating the sequences to 100 tokens and using a bidirectional `SimpleRNN` layer resulted in 80 percent classification accuracy. Although the prediction is slightly lower when compared to the previous bidirectional LSTM model (85.15 percent accuracy on the test dataset), the performance on these truncated sequences is much better than the performance we could achieve with a `SimpleRNN` on full-length movie reviews. As an optional exercise, you can verify this by using the two helper functions we have already defined. Try it with `max_seq_length=None` and set the `bidirectional` argument inside the `build_rnn_model()` helper function to `False`. (For your convenience, this code is available in the online materials of this book.)

Project two – character-level language modeling in TensorFlow

Language modeling is a fascinating application that enables machines to perform human language-related tasks, such as generating English sentences. One of the interesting studies in this area is *Generating Text with Recurrent Neural Networks*, *Ilya Sutskever*, *James Martens*, and *Geoffrey E. Hinton*, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, *2011*, `https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf`).

In the model that we will build now, the input is a text document, and our goal is to develop a model that can generate new text that is similar in style to the input document. Examples of such an input are a book or a computer program in a specific programming language.

In character-level language modeling, the input is broken down into a sequence of characters that are fed into our network one character at a time. The network will process each new character in conjunction with the memory of the previously seen characters to predict the next one. The following figure shows an example of character-level language modeling (note that EOS stands for "end of sequence"):


Data:
"Hello world!"
Break it into a sequence of characters
Input sequence:
'H' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' '!' EOS
Predicting next character:
'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' '!' EOS

We can break this implementation down into three separate steps: preparing the data, building the RNN model, and performing next-character prediction and sampling to generate new text.

Preprocessing the dataset

In this section, we will prepare the data for character-level language modeling.

To obtain the input data, visit the Project Gutenberg website at https://www.gutenberg.org/, which provides thousands of free e-books. For our example, you can download the book *The Mysterious Island*, by Jules Verne (published in 1874) in plain text format from http://www.gutenberg.org/files/1268/1268-0.txt.

Note that this link will take you directly to the download page. If you are using macOS or a Linux operating system, you can download the file with the following command in the terminal:

```
curl -O http://www.gutenberg.org/files/1268/1268-0.txt
```

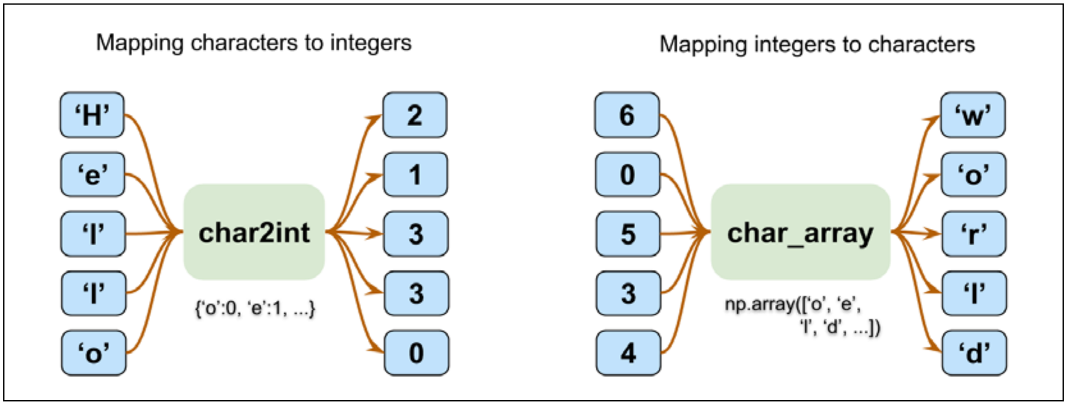
If this resource becomes unavailable in the future, a copy of this text is also included in this chapter's code directory in the book's code repository at https://github.com/rasbt/python-machine-learning-book-3rd-edition/code/ch16.

Once we have downloaded the dataset, we can read it into a Python session as plain text. Using the following code, we will read the text directly from the downloaded file and remove portions from the beginning and the end (these contain certain descriptions of the Gutenberg project). Then, we will create a Python variable, char_set, that represents the set of *unique* characters observed in this text:

```
>>> import numpy as np

>>> ## Reading and processing text
>>> with open('1268-0.txt', 'r') as fp:
```

```
...     text=fp.read()

>>> start_indx = text.find('THE MYSTERIOUS ISLAND')
>>> end_indx = text.find('End of the Project Gutenberg')
>>> text = text[start_indx:end_indx]
>>> char_set = set(text)
>>> print('Total Length:', len(text))
Total Length: 1112350

>>> print('Unique Characters:', len(char_set))
Unique Characters: 80
```

After downloading and preprocessing the text, we have a sequence consisting of 1,112,350 characters in total and 80 unique characters. However, most NN libraries and RNN implementations cannot deal with input data in string format, which is why we have to convert the text into a numeric format. To do this, we will create a simple Python dictionary that maps each character to an integer, `char2int`. We will also need a reverse mapping to convert the results of our model back to text. Although the reverse can be done using a dictionary that associates integer keys with character values, using a NumPy array and indexing the array to map indices to those unique characters is more efficient. The following figure shows an example of converting characters into integers and the reverse for the words `"Hello"` and `"world"`:



Building the dictionary to map characters to integers, and reverse mapping via indexing a NumPy array, as was shown in the previous figure, is as follows:

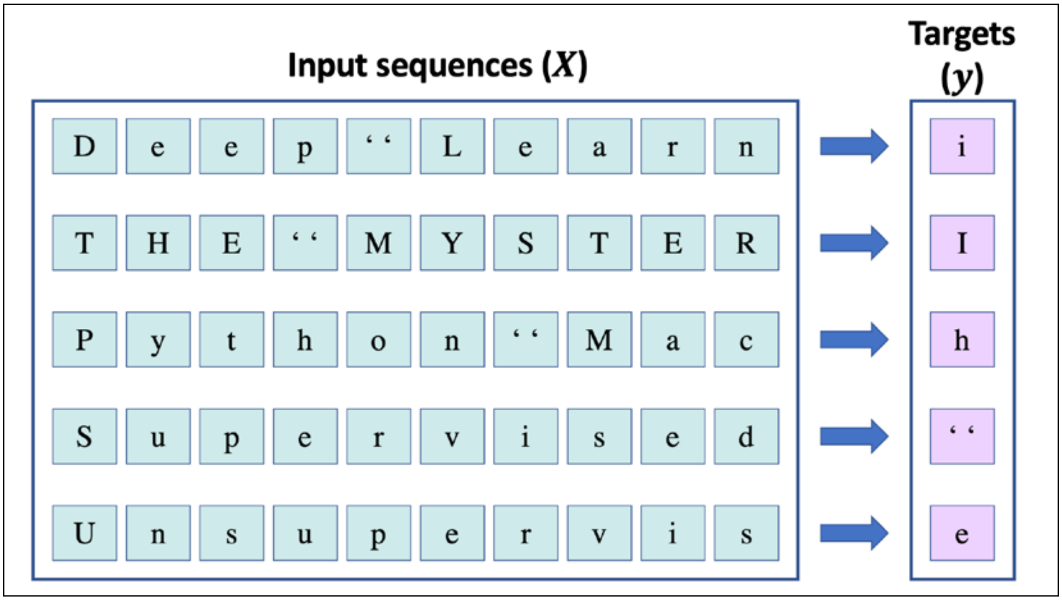
```
>>> chars_sorted = sorted(char_set)
>>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>>> char_array = np.array(chars_sorted)

>>> text_encoded = np.array(
...     [char2int[ch] for ch in text],
...     dtype=np.int32)

>>> print('Text encoded shape:', text_encoded.shape)
Text encoded shape: (1112350,)

>>> print(text[:15], '== Encoding ==>', text_encoded[:15])
>>> print(text_encoded[15:21], '== Reverse ==>',
...     ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS == Encoding ==> [44 32 29  1 37 48 43 44 29 42 33 39 45
43  1]
[33 43 36 25 38 28] == Reverse ==> ISLAND
```

The NumPy array `text_encoded` contains the encoded values for all the characters in the text. Now, we will create a TensorFlow dataset from this array:

```
>>> import tensorflow as tf

>>> ds_text_encoded = tf.data.Dataset.from_tensor_slices(
...                   text_encoded)
>>> for ex in ds_text_encoded.take(5):
...     print('{} -> {}'.format(ex.numpy(), char_array[ex.numpy()]))
44 -> T
32 -> H
29 -> E
1 ->
37 -> M
```

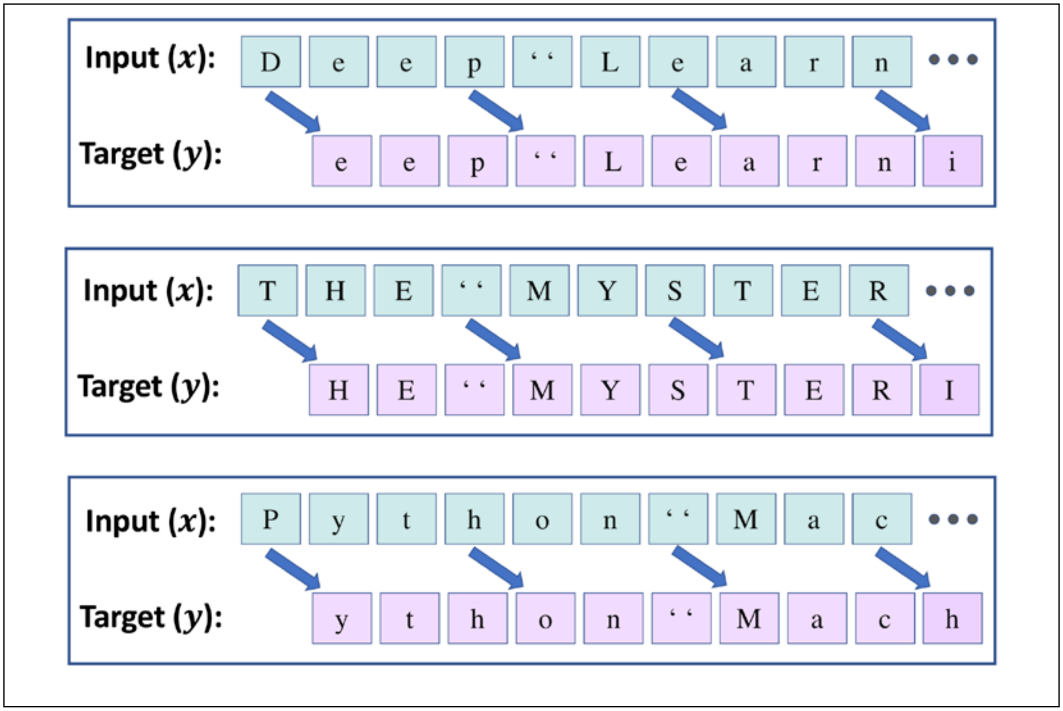
So far, we have created an iterable `Dataset` object for obtaining characters in the order they appear in the text. Now, let's step back and look at the big picture of what we are trying to do. For the text generation task, we can formulate the problem as a classification task.

Suppose we have a set of sequences of text characters that are incomplete, as shown in the following figure:


Input sequences (*X*)
Targets (*y*)
D e e p ' ' L e a r n → i
T H E ' ' M Y S T E R → I
P y t h o n ' ' M a c → h
S u p e r v i s e d → ' '
U n s u p e r v i s → e

In the previous figure, we can consider the sequences shown in the left-hand box to be the input. In order to generate new text, our goal is to design a model that can predict the next character of a given input sequence, where the input sequence represents an incomplete text. For example, after seeing "*Deep Learn*", the model should predict "*i*" as the next character. Given that we have 80 unique characters, this problem becomes a multiclass classification task.

Starting with a sequence of length 1 (that is, one single letter), we can iteratively generate new text based on this multiclass classification approach, as illustrated in the following figure:



To implement the text generation task in TensorFlow, let's first clip the sequence length to 40. This means that the input tensor, ***x***, consists of 40 tokens. In practice, the sequence length impacts the quality of the generated text. Longer sequences can result in more meaningful sentences. For shorter sequences, however, the model might focus on capturing individual words correctly, while ignoring the context for the most part. Although longer sequences usually result in more meaningful sentences, as mentioned, for long sequences, the RNN model will have problems capturing long-term dependencies. Thus, in practice, finding a sweet spot and good value for the sequence length is a hyperparameter optimization problem, which we have to evaluate empirically. Here, we are going to choose 40, as it offers a good tradeoff.

As you can see in the previous figure, the inputs, ***x***, and targets, ***y***, are offset by one character. Hence, we will split the text into chunks of size 41: the first 40 characters will form the input sequence, ***x***, and the last 40 elements will form the target sequence, ***y***.

We have already stored the entire encoded text in its original order in a `Dataset` object, `ds_text_encoded`. Using the techniques concerning transforming datasets that we already covered in this chapter (in the section *Preparing the movie review data*), can you think of a way to obtain the input, ***x***, and target, ***y***, as it was shown in the previous figure? The answer is very simple: we will first use the `batch()` method to create text chunks consisting of 41 characters each. This means that we will set `batch_size=41`. We will further get rid of the last batch if it is shorter than 41 characters. As a result, the new chunked dataset, named `ds_chunks`, will always contain sequences of size 41. The 41-character chunks will then be used to construct the sequence ***x*** (that is, the input), as well as the sequence ***y*** (that is, the target), both of which will have 40 elements. For instance, sequence ***x*** will consist of the elements with indices [0, 1, …, 39]. Furthermore, since sequence ***y*** will be shifted by one position with respect to ***x***, its corresponding indices will be [1, 2, …, 40]. Then, we will apply a transformation function using the `map()` method to separate the ***x*** and ***y*** sequences accordingly:

```
>>> seq_length = 40
>>> chunk_size = seq_length + 1

>>> ds_chunks = ds_text_encoded.batch(chunk_size,
...                                   drop_remainder=True)

>>> ## define the function for splitting x & y
>>> def split_input_target(chunk):
...     input_seq = chunk[:-1]
...     target_seq = chunk[1:]
...     return input_seq, target_seq

>>> ds_sequences = ds_chunks.map(split_input_target)
```

Let's take a look at some example sequences from this transformed dataset:

```
>>> for example in ds_sequences.take(2):
...     print(' Input (x): ',
...           repr(''.join(char_array[example[0].numpy()])))
...     print('Target (y): ',
...           repr(''.join(char_array[example[1].numpy()])))
...     print()

 Input (x):  'THE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced b'
```

```
Target (y):  'HE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by'

 Input (x):  ' Anthony Matonak, and Trevor Carlson\n\n\n\n'
Target (y):  'Anthony Matonak, and Trevor Carlson\n\n\n\n\n'
```

Finally, the last step in preparing the dataset is to divide this dataset into mini-batches. During the first preprocessing step to divide the dataset into batches, we created chunks of sentences. Each chunk represents one sentence, which corresponds to one training example. Now, we will shuffle the training examples and divide the inputs into mini-batches again; however, this time, each batch will contain multiple training examples:

```
>>> BATCH_SIZE = 64
>>> BUFFER_SIZE = 10000
>>> ds = ds_sequences.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Building a character-level RNN model

Now that the dataset is ready, building the model will be relatively straightforward. For code reusability, we will write a function, `build_model`, that defines an RNN model using the Keras `Sequential` class. Then, we can specify the training parameters and call that function to obtain an RNN model:

```
>>> def build_model(vocab_size, embedding_dim,rnn_units):
...     model = tf.keras.Sequential([
...         tf.keras.layers.Embedding(vocab_size, embedding_dim),
...         tf.keras.layers.LSTM(
...             rnn_units,
...             return_sequences=True),
...         tf.keras.layers.Dense(vocab_size)
...     ])
...     return model

>>> ## Setting the training parameters
>>> charset_size = len(char_array)
>>> embedding_dim = 256
>>> rnn_units = 512

>>> tf.random.set_seed(1)
>>> model = build_model(
...     vocab_size=charset_size,
...     embedding_dim=embedding_dim,
...     rnn_units=rnn_units)

>>> model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 256)         20480
_________________________________________________________________
lstm (LSTM)                  (None, None, 512)         1574912
_________________________________________________________________
dense (Dense)                (None, None, 80)          41040
=================================================================
Total params: 1,636,432
Trainable params: 1,636,432
Non-trainable params: 0
_________________________________________________________________
```

Notice that the LSTM layer in this model has the output shape `(None, None, 512)`, which means the output of LSTM is rank 3. The first dimension stands for the number of batches, the second dimension for the output sequence length, and the last dimension corresponds to the number of hidden units. The reason for having rank-3 output from the LSTM layer is because we have specified `return_sequences=True` when defining our LSTM layer. A fully connected layer (`Dense`) receives the output from the LSTM cell and computes the logits for each element of the output sequences. As a result, the final output of the model will be a rank-3 tensor as well.

Furthermore, we specified `activation=None` for the final fully connected layer. The reason for this is that we will need to have the logits as outputs of the model so that we can sample from the model predictions in order to generate new text. We will get to this sampling part later. For now, let's train the model:

```
>>> model.compile(
...     optimizer='adam',
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(
...         from_logits=True
...     ))

>>> model.fit(ds, epochs=20)

Epoch 1/20
424/424 [==============================] - 80s 189ms/step - loss:
2.3437
Epoch 2/20
424/424 [==============================] - 79s 187ms/step - loss:
1.7654
...
Epoch 20/20
```

```
424/424 [==============================] - 79s 187ms/step - loss:
1.0478
```

Now, we can evaluate the model to generate new text, starting with a given short string. In the next section, we will define a function to evaluate the trained model.

Evaluation phase – generating new text passages

The RNN model we trained in the previous section returns the logits of size 80 for each unique character. These logits can be readily converted to probabilities, via the softmax function, that a particular character will be encountered as the next character. To predict the next character in the sequence, we can simply select the element with the maximum logit value, which is equivalent to selecting the character with the highest probability. However, instead of always selecting the character with the highest likelihood, we want to (randomly) *sample* from the outputs; otherwise, the model will always produce the same text. TensorFlow already provides a function, `tf.random.categorical()`, which we can use to draw random samples from a categorical distribution. To see how this works, let's generate some random samples from three categories [0, 1, 2], with input logits [1, 1, 1].

```
>>> tf.random.set_seed(1)

>>> logits = [[1.0, 1.0, 1.0]]
>>> print('Probabilities:', tf.math.softmax(logits).numpy()[0])
Probabilities: [0.33333334 0.33333334 0.33333334]

>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
>>> tf.print(samples.numpy())
array([[0, 0, 1, 2, 0, 0, 0, 0, 1, 0]])
```

As you can see, with the given logits, the categories have the same probabilities (that is, equiprobable categories). Therefore, if we use a large sample size ($num\_samples \rightarrow \infty$), we would expect the number of occurrences of each category to reach ≈ 1/3 of the sample size. If we change the logits to [1, 1, 3], then we would expect to observe more occurrences for category 2 (when a very large number of examples are drawn from this distribution):

```
>>> tf.random.set_seed(1)

>>> logits = [[1.0, 1.0, 3.0]]
>>> print('Probabilities: ', tf.math.softmax(logits).numpy()[0])
Probabilities: [0.10650698 0.10650698 0.78698605]

>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
```

```
>>> tf.print(samples.numpy())
array([[2, 0, 2, 2, 2, 0, 1, 2, 2, 0]])
```

Using `tf.random.categorical`, we can generate examples based on the logits computed by our model. We define a function, `sample()`, that receives a short starting string, `starting_str`, and generate a new string, `generated_str`, which is initially set to the input string. Then, a string of size `max_input_length` is taken from the end of `generated_str` and encoded to a sequence of integers, `encoded_input`. The `encoded_input` is passed to the RNN model to compute the logits. Note that the output from the RNN model is a sequence of logits with the same length as the input sequence, since we specified `return_sequences=True` for the last recurrent layer of our RNN model. Therefore, each element in the output of the RNN model represents the logits (here, a vector of size 80, which is the total number of characters) for the next character after observing the input sequence by the model.

Here, we only use the last element of the output `logits` (that is, $\boldsymbol{o}^{(T)}$), which is passed to the `tf.random.categorical()` function to generate a new sample. This new sample is converted to a character, which is then appended to the end of the generated string, `generated_text`, increasing its length by 1. Then, this process is repeated, taking the last `max_input_length` number of characters from the end of the `generated_str`, and using that to generate a new character until the length of the generated string reaches the desired value. The process of consuming the generated sequence as input for generating new elements is called *auto-regression*.

Returning sequences as output

You may wonder why we use `return_sequences=True` when we only use the last character to sample a new character and ignore the rest of the output. While this question makes perfect sense, you should not forget that we used the entire output sequence for training. The loss is computed based on each prediction in the output and not just the last one.

The code for the `sample()` function is as follows:

```
>>> def sample(model, starting_str,
...            len_generated_text=500,
...            max_input_length=40,
...            scale_factor=1.0):
...     encoded_input = [char2int[s] for s in starting_str]
...     encoded_input = tf.reshape(encoded_input, (1, -1))
...
...     generated_str = starting_str
...
```

```
...     model.reset_states()
...     for i in range(len_generated_text):
...         logits = model(encoded_input)
...         logits = tf.squeeze(logits, 0)
...
...         scaled_logits = logits * scale_factor
...         new_char_indx = tf.random.categorical(
...             scaled_logits, num_samples=1)
...
...         new_char_indx = tf.squeeze(new_char_indx)[-1].numpy()
...
...         generated_str += str(char_array[new_char_indx])
...
...         new_char_indx = tf.expand_dims([new_char_indx], 0)
...         encoded_input = tf.concat(
...             [encoded_input, new_char_indx],
...             axis=1)
...         encoded_input = encoded_input[:, -max_input_length:]
...
...     return generated_str
```

Let's now generate some new text:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island'))
```

```
The island is probable that the view of the vegetable discharge on
unexplainst felt, a thore, did not
refrain it existing to the greatest
possing bain and production, for a hundred streamled
established some branches of the
holizontal direction. It was there is all ready, from one things from
contention of the Pacific
acid, and
according to an occurry so
summ on the rooms. When numbered the prud Spilett received an
exceppering from their head, and by went inhabited.

"What are the most abundance a report
```

As you can see, the model generates mostly correct words, and, in some cases, the sentences are partially meaningful. You can further tune the training parameters, such as the length of input sequences for training, the model architecture, and sampling parameters (such as `max_input_length`).

Furthermore, in order to control the predictability of the generated samples (that is, generating text following the learned patterns from the training text versus adding more randomness), the logits computed by the RNN model can be scaled before being passed to `tf.random.categorical()` for sampling. The scaling factor, α, can be interpreted as the inverse of the temperature in physics. Higher temperatures result in more randomness versus more predictable behavior at lower temperatures. By scaling the logits with $\alpha < 1$, the probabilities computed by the softmax function become more uniform, as shown in the following code:

```
>>> logits = np.array([[1.0, 1.0, 3.0]])

>>> print('Probabilities before scaling:        ',
...       tf.math.softmax(logits).numpy()[0])

>>> print('Probabilities after scaling with 0.5:',
...       tf.math.softmax(0.5*logits).numpy()[0])

>>> print('Probabilities after scaling with 0.1:',
...       tf.math.softmax(0.1*logits).numpy()[0])

Probabilities before scaling:         [0.10650698 0.10650698
0.78698604]
Probabilities after scaling with 0.5: [0.21194156 0.21194156
0.57611688]
Probabilities after scaling with 0.1: [0.31042377 0.31042377
0.37915245]
```

As you can see, scaling the logits by $\alpha = 0.1$ results in near-uniform probabilities [0.31, 0.31, 0.38]. Now, we can compare the generated text with $\alpha = 2.0$ and $\alpha = 0.5$, as shown in the following points:

- $\alpha = 2.0$ ➔ more predictable:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=2.0))
The island spoke of heavy torn into the island from the sea.

The noise of the inhabitants of the island was to be feared that
the colonists had come a project with a straight be put to the
bank of the island was the surface of the lake and sulphuric acid,
and several supply of her animals. The first stranger carried a
sort of accessible to break these screen barrels to their distance
from the palisade.

"The first huntil," said the reporter, "and his companions the
reporter extended to build a few days a
```

- $\alpha = 0.5$ ➔ more randomness:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=0.5))
The island
glissed in
ascercicedly useful? loigeh, Cyrus,
Spileots," henseporvemented
House to a left
the centlic moment. Tonsense craw.

Pencrular ed/ of times," tading had coflently often above anzand?"

"Wat;" then:y."


Ardivify he acpearly, howcovered--he hassime; however, fenquests
hen adgents!'.? Let us Neg eqiAl?.

GencNal, my surved thirtyin" ou; is Harding; treuths. Osew
apartarned. "N,
the poltuge of about-but durired with purteg.

Chappes wason!

Fears," returned Spilett; "if
you tear 8t trung
```

The results show that scaling the logits with $\alpha = 0.5$ (increasing the temperature) generates more random text. There is a tradeoff between the novelty of the generated text and its correctness.

In this section, we worked with character-level text generation, which is a sequence-to-sequence (seq2seq) modeling task. While this example may not be very useful by itself, it is easy to think of several useful applications for these types of models; for example, a similar RNN model can be trained as a chatbot to assist users with simple queries.

Understanding language with the Transformer model

In this chapter, we solved two sequence modeling problems using RNN-based NNs. However, a new architecture has recently emerged that has been shown to outperform the RNN-based seq2seq models in several NLP tasks.

It is called the **Transformer** architecture, capable of modeling global dependencies between input and output sequences, and was introduced in 2017 by Ashish Vaswani, et. al., in the NeurIPS paper *Attention Is All You Need* (available online at `http://papers.nips.cc/paper/7181-attention-is-all-you-need`). The Transformer architecture is based on a concept called **attention**, and more specifically, the **self-attention mechanism**. Let's consider the sentiment analysis task that we covered earlier in this chapter. In this case, using the attention mechanism would mean that our model would be able to learn to focus on the parts of an input sequence that are more relevant to the sentiment.

Understanding the self-attention mechanism

This section will explain the *self-attention mechanism* and how it helps a Transformer model to focus on important parts of a sequence for NLP. The first subsection will cover a very basic form of self-attention to illustrate the overall idea behind learning text representations. Then, we will add different weight parameters so that we arrive at the self-attention mechanism that is commonly used in Transformer models.

A basic version of self-attention

To introduce the basic idea behind self-attention, let's assume we have an input sequence of length *T*, $\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(T)}$, as well as an output sequence, $\boldsymbol{o}^{(0)}, \boldsymbol{o}^{(1)}, \dots, \boldsymbol{o}^{(T)}$. Each element of these sequences, $\boldsymbol{x}^{(t)}$ and $\boldsymbol{o}^{(t)}$, are vectors of size *d* (that is, $\boldsymbol{x}^{(t)} \in R^d$). Then, for a seq2seq task, the goal of self-attention is to model the dependencies of each element in the output sequence to the input elements. In order to achieve this, attention mechanisms are composed of three stages. Firstly, we derive importance weights based on the similarity between the current element and all other elements in the sequence. Secondly, we normalize the weights, which usually involves the use of the already familiar softmax function. Thirdly, we use these weights in combination with the corresponding sequence elements in order to compute the attention value.

More formally, the output of self-attention is the weighted sum of all input sequences. For instance, for the *i*th input element, the corresponding output value is computed as follows:

$$\boldsymbol{o}^{(i)} = \sum_{j=0}^{T} \boldsymbol{W}_{ij} \boldsymbol{x}^{(j)}$$

Here, the weights, W_{ij}, are computed based on the similarity between the current input element, $\boldsymbol{x}^{(i)}$, and all other elements in the input sequence. More concretely, this similarity is computed as the dot product between the current input element, $\boldsymbol{x}^{(i)}$, and another element in the input sequence, $\boldsymbol{x}^{(j)}$:

$$\omega_{ij} = \boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(j)}$$

After computing these similarity-based weights for the ith input and all inputs in the sequence ($\boldsymbol{x}^{(i)}$ to $\boldsymbol{x}^{(T)}$), the "raw" weights (ω_{i0} to ω_{iT}) are then normalized using the familiar softmax function, as follows:

$$W_{ij} = \frac{\exp(\omega_{ij})}{\sum_{j=0}^{T} \exp(\omega_{ij})} = softmax\left(\left[\omega_{ij}\right]_{j=0\ldots T}\right)$$

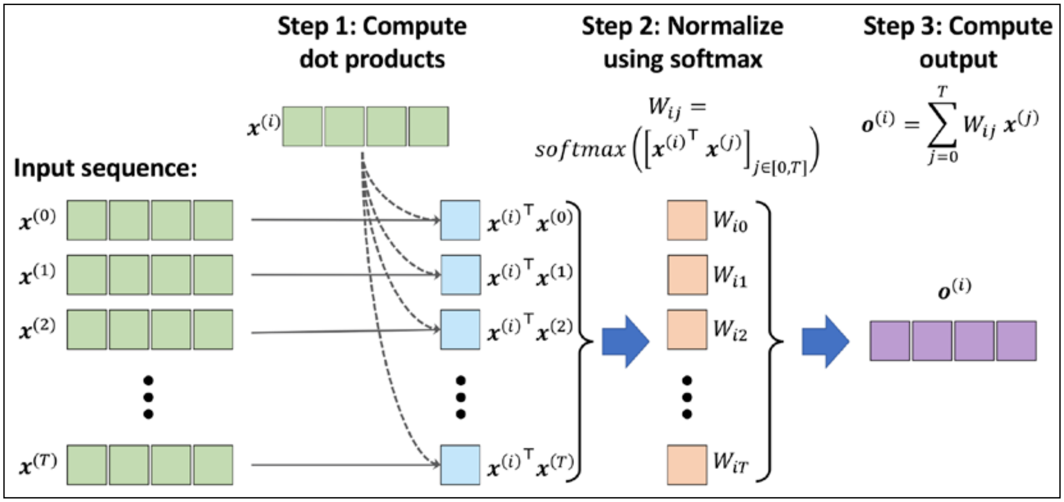
Notice that as a consequence of applying the softmax function, the weights will sum to 1 after this normalization, that is,

$$\sum_{j=0}^{T} W_{ij} = 1$$

To recap, let's summarize the three main steps behind the self-attention operation:

1. For a given input element, $\boldsymbol{x}^{(i)}$, and each jth element in the range [0, T], compute the dot product, $\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(j)}$
2. Obtain the weight, W_{ij}, by normalizing the dot products using the softmax function
3. Compute the output, $\boldsymbol{o}^{(i)}$, as the weighted sum over the entire input sequence: $\boldsymbol{o}^{(i)} = \sum_{j=0}^{T} W_{ij}\, \boldsymbol{x}^{(j)}$

These steps are further illustrated in the following figure:


Step 1: Compute dot products
Step 2: Normalize using softmax
Step 3: Compute output
$\boldsymbol{x}^{(i)}$
$W_{ij} = softmax\left(\left[\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(j)}\right]_{j\in[0,T]}\right)$
$\boldsymbol{o}^{(i)} = \sum_{j=0}^{T} W_{ij}\, \boldsymbol{x}^{(j)}$
Input sequence:
$\boldsymbol{x}^{(0)}$
$\boldsymbol{x}^{(1)}$
$\boldsymbol{x}^{(2)}$
$\boldsymbol{x}^{(T)}$
$\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(0)}$
$\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(1)}$
$\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(2)}$
$\boldsymbol{x}^{(i)^\top} \boldsymbol{x}^{(T)}$
W_{i0}
W_{i1}
W_{i2}
W_{iT}
$\boldsymbol{o}^{(i)}$

Parameterizing the self-attention mechanism with query, key, and value weights

Now that you have been introduced to the basic concept behind self-attention, this subsection summarizes the more advanced self-attention mechanism that is used in the Transformer model. Note that in the previous subsection, we didn't involve any learnable parameters when computing the outputs. Hence, if we want to learn a language model and want to change the attention values to optimize an objective, such as minimizing the classification error, we will need to change the word embeddings (that is, input vectors) that underlie each input element, $\boldsymbol{x}^{(i)}$. In other words, using the previously introduced basic self-attention mechanism, the Transformer model is rather limited with regard to how it can update or change the attention values during model optimization for a given sequence. To make the self-attention mechanism more flexible and amenable to model optimization, we will introduce three additional weight matrices that can be fit as model parameters during model training. We denote these three weight matrices as $\boldsymbol{U}_q$, $\boldsymbol{U}_k$, and $\boldsymbol{U}_v$. They are used to project the inputs into *query*, *key*, and *value* sequence elements:

- Query sequence: $\boldsymbol{q}^{(i)} = \boldsymbol{U}_q\, \boldsymbol{x}^{(i)}$ for $i \in [0, T]$,
- Key sequence: $\boldsymbol{k}^{(i)} = \boldsymbol{U}_k\, \boldsymbol{x}^{(i)}$ for $i \in [0, T]$,
- Value sequence: $\boldsymbol{v}^{(i)} = \boldsymbol{U}_v\, \boldsymbol{x}^{(i)}$ for $i \in [0, T]$

Here, both $\boldsymbol{q}^{(i)}$ and $\boldsymbol{k}^{(i)}$ are vectors of size d_k. Therefore, the projection matrices $\boldsymbol{U}_q$ and $\boldsymbol{U}_k$ have the shape $d_k \times d$, while $\boldsymbol{U}_v$ has the shape $d_v \times d$. For simplicity, we can design these vectors to have the same shape, for example, using $m = d_k = d_v$. Now, instead of computing the unnormalized weight as the pairwise dot product between the given input sequence element, $\boldsymbol{x}^{(i)}$, and the *j*th sequence element, $\boldsymbol{x}^{(j)}$, we can compute the dot product between the query and key:

$$\omega_{ij} = \boldsymbol{q}^{(i)^{\top}} \boldsymbol{k}^{(j)}$$

We can then further use m, or, more precisely, $1/\sqrt{m}$, to scale ω_{ij} before normalizing it via the softmax function, as follows:

$$W_{ij} = softmax\left(\frac{\omega_{ij}}{\sqrt{m}}\right)$$

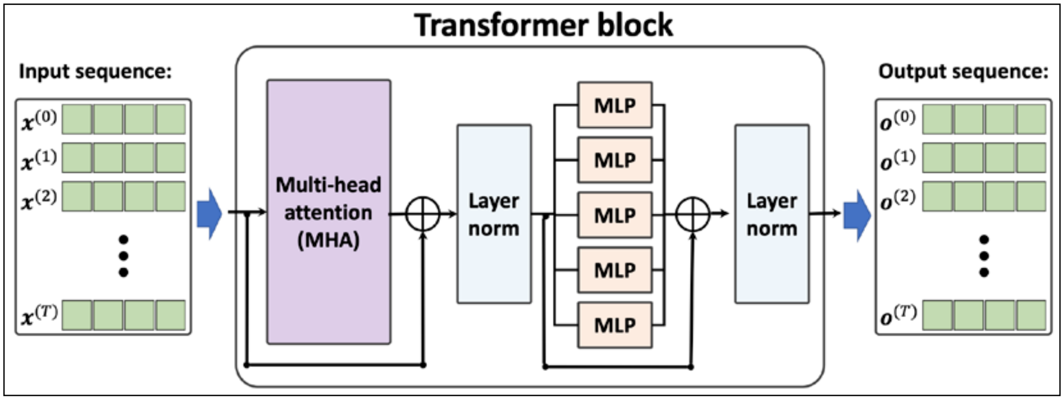
Note that scaling ω_{ij} by $1/\sqrt{m}$ will ensure that the Euclidean length of the weight vectors will be approximately in the same range.

Multi-head attention and the Transformer block

Another trick that greatly improves the discriminatory power of the self-attention mechanism is **multi-head attention** (**MHA**), which combines multiple self-attention operations together. In this case, each self-attention mechanism is called a *head*, which can be computed in parallel. Using *r* parallel heads, each head results in a vector, $\boldsymbol{h}$, of size m. These vectors are then concatenated to obtain a vector, $\boldsymbol{z}$, with the shape $r \times m$. Finally, the concatenated vector is projected using the output matrix $\boldsymbol{W}^o$ to obtain the final output, as follows:

$$\boldsymbol{o}^{(i)} = W_{ij}^o \, \boldsymbol{z}$$

The architecture of a Transformer block is shown in the following figure:



Notice that in the Transformer architecture shown in the previous figure, we added two additional components that we haven't discussed yet. One of these components is the *residual connection*, which adds the output from a layer (or even a group of layers) to its input, that is, $\boldsymbol{x}$ + *layer*($\boldsymbol{x}$). The block consisting of a layer (or multiple layers) with such a residual connection is called a *residual block*. The Transformer block shown in the previous figure has two residual blocks.

The other new component is *layer normalization*, which is denoted in the previous figure as "Layer norm." There is a family of normalization layers including batch normalization, which we will cover in *Chapter 17, Generative Adversarial Networks for Synthesizing New Data*. For now, you can think of layer normalization as a fancy or more advanced way of normalizing or scaling the NN inputs and activations in each layer.

Returning to the illustration of the Transformer model in the previous figure, let's now discuss how this model works. First, the input sequence is passed to the MHA layers, which is based on the self-attention mechanism that we discussed earlier. In addition, the input sequences are added to the output of the MHA layers via the residual connections—this ensures that the earlier layers will receive sufficient gradient signals during training, which is a common trick that is used to improve training speed and convergence. If you are interested, you can read more about the concept behind residual connections in the research article *Deep Residual Learning for Image Recognition*, by *Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun*, which is freely available at `http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html`.

After the input sequences are added to the output of the MHA layers, the outputs are normalized via layer normalization. These normalized signals then go through a series of MLP (that is, fully connected) layers, which also have a residual connection. Finally, the output from the residual block is normalized again and returned as the output sequence, which can be used for sequence classification or sequence generation.

Instructions for implementing and training Transformer models were omitted to conserve space. However, the interested reader can find an excellent implementation and walk-through in the official TensorFlow documentation at `https://www.tensorflow.org/tutorials/text/transformer`.

Summary

In this chapter, you first learned about the properties of sequences that make them different to other types of data, such as structured data or images. We then covered the foundations of RNNs for sequence modeling. You learned how a basic RNN model works and discussed its limitations with regard to capturing long-term dependencies in sequence data. Next, we covered LSTM cells, which consist of a gating mechanism to reduce the effect of exploding and vanishing gradient problems, which are common in basic RNN models.

After discussing the main concepts behind RNNs, we implemented several RNN models with different recurrent layers using the Keras API. In particular, we implemented an RNN model for sentiment analysis, as well as an RNN model for generating text. Finally, we covered the Transformer model, which leverages the self-attention mechanism in order to focus on the relevant parts of a sequence.

In the next chapter, you will learn about generative models and, in particular, **generative adversarial networks** (**GANs**), which have shown remarkable results in the computer vision community for various vision tasks.

17

Generative Adversarial Networks for Synthesizing New Data

In the previous chapter, we focused on **recurrent neural networks** for modeling sequences. In this chapter, we will explore **generative adversarial networks** (**GANs**) and see their application in synthesizing new data samples. GANs are considered to be the most important breakthrough in deep learning, allowing computers to generate new data (such as new images).

In this chapter, we will cover the following topics:

- Introducing generative models for synthesizing new data
- Autoencoders, **variational autoencoders** (**VAEs**), and their relationship to GANs
- Understanding the building blocks of GANs
- Implementing a simple GAN model to generate handwritten digits
- Understanding transposed convolution and **batch normalization** (**BatchNorm** or **BN**)
- Improving GANs: deep convolutional GANs and GANs using the Wasserstein distance

Introducing generative adversarial networks

Let's first look at the foundations of GAN models. The overall objective of a GAN is to synthesize new data that has the same distribution as its training dataset. Therefore, GANs, in their original form, are considered to be in the unsupervised learning category of machine learning tasks, since no labeled data is required. It is worth noting, however, that extensions made to the original GAN can lie in both semi-supervised and supervised tasks.

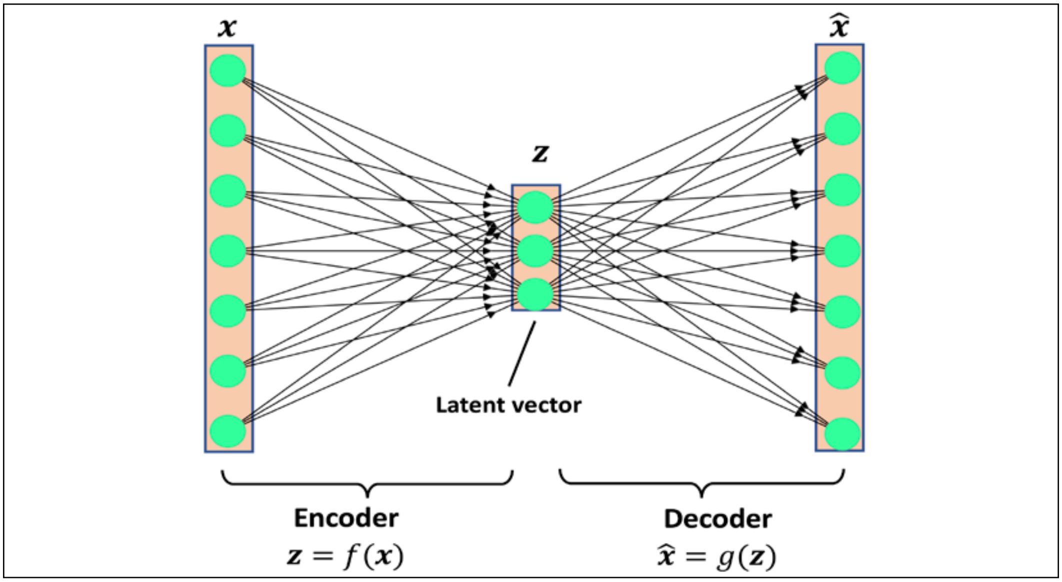
The general GAN concept was first proposed in 2014 by Ian Goodfellow and his colleagues as a method for synthesizing new images using deep neural networks (NNs) (Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., *Generative Adversarial Nets*, in *Advances in Neural Information Processing Systems*, pp. 2672-2680, 2014). While the initial GAN architecture proposed in this paper was based on fully connected layers, similar to multilayer perceptron architectures, and trained to generate low-resolution MNIST-like handwritten digits, it served more as a proof of concept to demonstrate the feasibility of this new approach.

However, since its introduction, the original authors, as well as many other researchers, have proposed numerous improvements and various applications in different fields of engineering and science; for example, in computer vision, GANs are used for image-to-image translation (learning how to map an input image to an output image), image super-resolution (making a high-resolution image from a low-resolution version), image inpainting (learning how to reconstruct the missing parts of an image), and many more applications. For instance, recent advances in GAN research have led to models that are able to generate new, high-resolution face images. Examples of such high-resolution images can be found on `https://www.thispersondoesnotexist.com/`, which showcases synthetic face images generated by a GAN.

Starting with autoencoders

Before we discuss how GANs work, we will first start with autoencoders, which can compress and decompress training data. While standard autoencoders cannot generate new data, understanding their function will help you to navigate GANs in the next section.

Autoencoders are composed of two networks concatenated together: an **encoder** network and a **decoder** network. The encoder network receives a d-dimensional input feature vector associated with example $\boldsymbol{x}$ (that is, $\boldsymbol{x} \in R^d$) and encodes it into a p-dimensional vector, $\boldsymbol{z}$ (that is, $\boldsymbol{z} \in R^p$). In other words, the role of the encoder is to learn how to model the function $\boldsymbol{z} = f(\boldsymbol{x})$. The encoded vector, $\boldsymbol{z}$, is also called the latent vector, or the latent feature representation. Typically, the dimensionality of the latent vector is less than that of the input examples; in other words, $p < d$. Hence, we can say that the encoder acts as a data compression function. Then, the decoder decompresses $\hat{\boldsymbol{x}}$ from the lower-dimensional latent vector, $\boldsymbol{z}$, where we can think of the decoder as a function, $\hat{\boldsymbol{x}} = g(\boldsymbol{z})$. A simple autoencoder architecture is shown in the following figure, where the encoder and decoder parts consist of only one fully connected layer each:



The connection between autoencoders and dimensionality reduction

In *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned about dimensionality reduction techniques, such as principal component analysis (PCA) and linear discriminant analysis (LDA). Autoencoders can be used as a dimensionality reduction technique as well. In fact, when there is no nonlinearity in either of the two subnetworks (encoder and decoder), then the autoencoder approach is *almost identical* to PCA.

In this case, if we assume the weights of a single-layer encoder (no hidden layer and no nonlinear activation function) are denoted by the matrix $\boldsymbol{U}$, then the encoder models $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$. Similarly, a single-layer linear decoder models $\hat{\boldsymbol{x}} = \boldsymbol{U}\boldsymbol{z}$. Putting these two components together, we have $\hat{\boldsymbol{x}} = \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{x}$. This is exactly what PCA does, with the exception that PCA has an additional orthonormal constraint: $\boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}_{n\times n}$.

While the previous figure depicts an autoencoder without hidden layers within the encoder and decoder, we can, of course, add multiple hidden layers with nonlinearities (as in a multilayer NN) to construct a deep autoencoder that can learn more effective data compression and reconstruction functions. Also, note that the autoencoder mentioned in this section uses fully connected layers. When we work with images, however, we can replace the fully connected layers with convolutional layers, as you learned in *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*.

Other types of autoencoders based on the size of latent space

As previously mentioned, the dimensionality of an autoencoder's latent space is typically lower than the dimensionality of the inputs ($p < d$), which makes autoencoders suitable for dimensionality reduction. For this reason, the latent vector is also often referred to as the "bottleneck," and this particular configuration of an autoencoder is also called *undercomplete*. However, there is a different category of autoencoders, called *overcomplete*, where the dimensionality of the latent vector, $\boldsymbol{z}$, is, in fact, greater than the dimensionality of the input examples ($p > d$).

When training an overcomplete autoencoder, there is a trivial solution where the encoder and the decoder can simply learn to copy (memorize) the input features to their output layer. Obviously, this solution is not very useful. However, with some modifications to the training procedure, overcomplete autoencoders can be used for *noise reduction*.

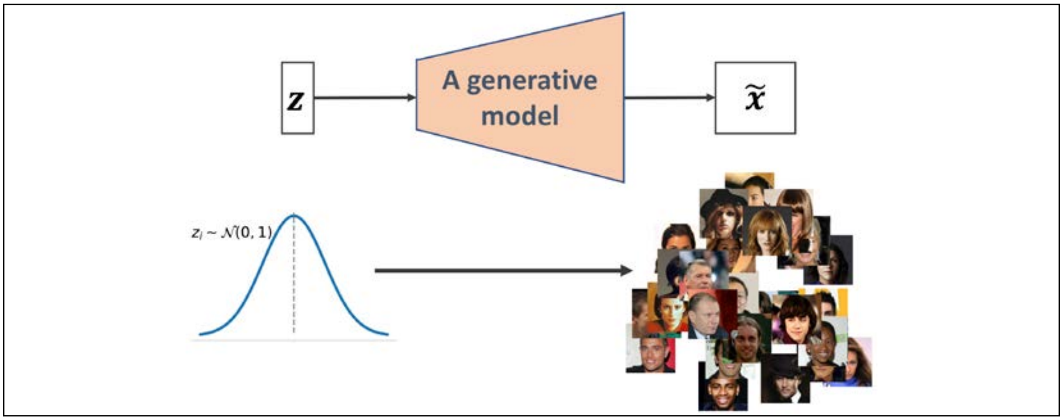
In this case, during training, random noise, $\boldsymbol{\epsilon}$, is added to the input examples and the network learns to reconstruct the clean example, $\boldsymbol{x}$, from the noisy signal, $\boldsymbol{x} + \boldsymbol{\epsilon}$. Then, at evaluation time, we provide the new examples that are naturally noisy (that is, noise is already present such that no additional artificial noise, $\boldsymbol{\epsilon}$, is added) in order to remove the existing noise from these examples. This particular autoencoder architecture and training method is referred to as a *denoising autoencoder*.

If you are interested, you can learn more about it in the research article *Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion* by Vincent et al., which is freely available at `http://www.jmlr.org/papers/v11/vincent10a.html`.

Generative models for synthesizing new data

Autoencoders are deterministic models, which means that after an autoencoder is trained, given an input, $\boldsymbol{x}$, it will be able to reconstruct the input from its compressed version in a lower-dimensional space. Therefore, it cannot generate new data beyond reconstructing its input through the transformation of the compressed representation.

A generative model, on the other hand, can generate a new example, $\tilde{\boldsymbol{x}}$, from a random vector, $\boldsymbol{z}$ (corresponding to the latent representation). A schematic representation of a generative model is shown in the following figure. The random vector, $\boldsymbol{z}$, comes from a simple distribution with fully known characteristics, so we can easily sample from such a distribution. For example, each element of $\boldsymbol{z}$ may come from the uniform distribution in the range [–1, 1] (for which we write $z_i \sim Uniform(-1, 1)$) or from a standard normal distribution (in which case, we write $z_i \sim Normal(\mu = 0, \sigma^2 = 1)$).



As we have shifted our attention from autoencoders to generative models, you may have noticed that the decoder component of an autoencoder has some similarities with a generative model. In particular, they both receive a latent vector, $\boldsymbol{z}$, as input and return an output in the same space as $\boldsymbol{x}$. (For the autoencoder, $\hat{\boldsymbol{x}}$ is the reconstruction of an input, $\boldsymbol{x}$, and for the generative model, $\tilde{\boldsymbol{x}}$ is a synthesized sample.)

However, the major difference between the two is that we do not know the distribution of $\boldsymbol{z}$ in the autoencoder, while in a generative model, the distribution of $\boldsymbol{z}$ is fully characterizable. It is possible to generalize an autoencoder into a generative model, though. One approach is **VAEs**.

In a VAE receiving an input example, $\boldsymbol{x}$, the encoder network is modified in such a way that it computes two moments of the distribution of the latent vector: the mean, $\boldsymbol{\mu}$, and variance, $\boldsymbol{\sigma}^2$. During the training of a VAE, the network is forced to match these moments with those of a standard normal distribution (that is, zero mean and unit variance). Then, after the VAE model is trained, the encoder is discarded, and we can use the decoder network to generate new examples, $\widetilde{\boldsymbol{x}}$, by feeding random $\boldsymbol{z}$ vectors from the "learned" Gaussian distribution.

Besides VAEs, there are other types of generative models, for example, *autoregressive models* and *normalizing flow models*. However, in this chapter, we are only going to focus on GAN models, which are among the most recent and most popular types of generative models in deep learning.

What is a generative model?

Note that generative models are traditionally defined as algorithms that model data input distributions, $p(\boldsymbol{x})$, or the joint distributions of the input data and associated targets, $p(\boldsymbol{x}, y)$. By definition, these models are also capable of sampling from some feature, $\boldsymbol{x}_i$, conditioned on another feature, $\boldsymbol{x}_j$, which is known as *conditional inference*. In the context of deep learning, however, the term *generative model* is typically used to refer to models that generate realistic-looking data. This means that we can sample from input distributions, $p(\boldsymbol{x})$, but we are not necessarily able to perform conditional inference.

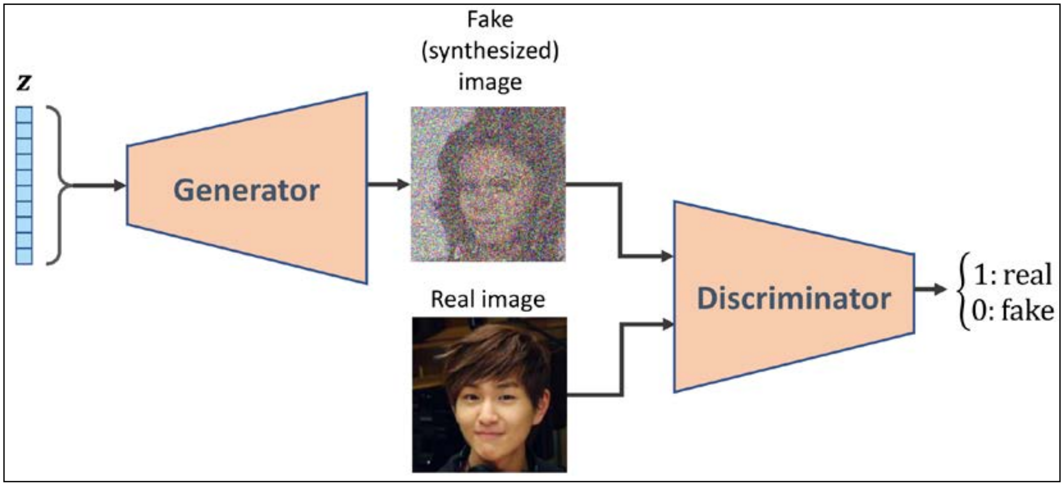
Generating new samples with GANs

To understand what GANs do in a nutshell, let's first assume we have a network that receives a random vector, $\boldsymbol{z}$, sampled from a known distribution and generates an output image, $\boldsymbol{x}$. We will call this network **generator** (G) and use the notation $\widetilde{\boldsymbol{x}} = G(\boldsymbol{z})$ to refer to the generated output. Assume our goal is to generate some images, for example, face images, images of buildings, images of animals, or even handwritten digits such as MNIST.

As always, we will initialize this network with random weights. Therefore, the first output images, before these weights are adjusted, will look like white noise. Now, imagine there is a function that can assess the quality of images (let's call it an *assessor function*).

If such a function exists, we can use the feedback from that function to tell our generator network how to adjust its weights in order to improve the quality of the generated images. This way, we can train the generator based on the feedback from that assessor function, such that the generator learns to improve its output toward producing realistic-looking images.

While an assessor function, as described in the previous paragraph, would make the image generation task very easy, the question is whether such a universal function to assess the quality of images exists and, if so, how it is defined. Obviously, as humans, we can easily assess the quality of output images when we observe the outputs of the network; although, we cannot (yet) backpropagate the result from our brain to the network. Now, if our brain can assess the quality of synthesized images, can we design an NN model to do the same thing? In fact, that's the general idea of a GAN. As shown in the following figure, a GAN model consists of an additional NN called **discriminator** (*D*), which is a classifier that learns to detect a synthesized image, $\tilde{\boldsymbol{x}}$, from a real image, $\boldsymbol{x}$:



In a GAN model, the two networks, generator and discriminator, are trained together. At first, after initializing the model weights, the generator creates images that do not look realistic. Similarly, the discriminator does a poor job of distinguishing between real images and images synthesized by the generator. But over time (that is, through training), both networks become better as they interact with each other. In fact, the two networks play an adversarial game, where the generator learns to improve its output to be able to fool the discriminator. At the same time, the discriminator becomes better at detecting the synthesized images.

Understanding the loss functions of the generator and discriminator networks in a GAN model

The objective function of GANs, as described in the original paper *Generative Adversarial Nets* by Goodfellow et al. (https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf), is as follows:

$$V\left(\theta^{(D)}, \theta^{(G)}\right) = E_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$$

Here, $V\left(\theta^{(D)}, \theta^{(G)}\right)$ is called the *value function*, which can be interpreted as a payoff: we want to maximize its value with respect to the discriminator (*D*), while minimizing its value with respect to the generator (G), that is, $\min_G \max_D V(\theta^{(D)}, \theta^{(G)})$. *D*($\boldsymbol{x}$) is the probability that indicates whether the input example, $\boldsymbol{x}$, is real or fake (that is, generated). The expression $E_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})]$ refers to the expected value of the quantity in brackets with respect to the examples from the data distribution (distribution of the real examples); $E_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$ refers to the expected value of the quantity with respect to the distribution of the input, $\boldsymbol{z}$, vectors.

One training step of a GAN model with such a value function requires two optimization steps: (1) maximizing the payoff for the discriminator and (2) minimizing the payoff for the generator. A practical way of training GANs is to alternate between these two optimization steps: (1) fix (freeze) the parameters of one network and optimize the weights of the other one, and (2) fix the second network and optimize the first one. This process should be repeated at each training iteration. Let's assume that the generator network is fixed, and we want to optimize the discriminator. Both terms in the value function $V\left(\theta^{(D)}, \theta^{(G)}\right)$ contribute to optimizing the discriminator, where the first term corresponds to the loss associated with the real examples, and the second term is the loss for the fake examples. Therefore, when *G* is fixed, our objective is to *maximize* $V\left(\theta^{(D)}, \theta^{(G)}\right)$, which means making the discriminator better at distinguishing between real and generated images.

After optimizing the discriminator using the loss terms for real and fake samples, we then fix the discriminator and optimize the generator. In this case, only the second term in $V\left(\theta^{(D)}, \theta^{(G)}\right)$ contributes to the gradients of the generator. As a result, when D is fixed, our objective is to *minimize* $V\left(\theta^{(D)}, \theta^{(G)}\right)$, which can be written as $\min_{G} E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$. As was mentioned in the original GAN paper by Goodfellow et al., this function, $\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$, suffers from vanishing gradients in the early training stages. The reason for this is that the outputs, $G(\boldsymbol{z})$, early in the learning process, look nothing like real examples, and therefore $D(G(\boldsymbol{z}))$ will be close to zero with high confidence. This phenomenon is called *saturation*. To resolve this issue, we can reformulate the minimization objective, $\min_{G} E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$, by rewriting it as $\max_{G} E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(D\left(G(\boldsymbol{z})\right)\right)\right]$.

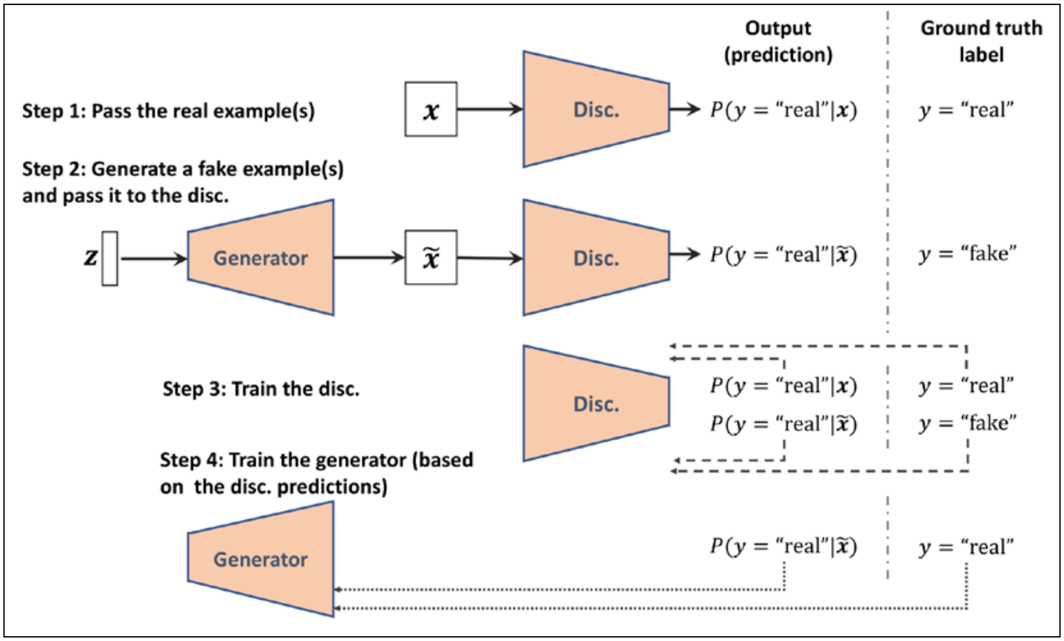
This replacement means that for training the generator, we can swap the labels of real and fake examples and carry out a regular function minimization. In other words, even though the examples synthesized by the generator are fake and are therefore labeled 0, we can flip the labels by assigning label 1 to these examples, and *minimize* the binary cross-entropy loss with these new labels instead of maximizing $\max_{G} E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(D\left(G(\boldsymbol{z})\right)\right)\right]$.

Now that we have covered the general optimization procedure for training GAN models, let's explore the various data labels that we can use when training GANs. Given that the discriminator is a binary classifier (the class labels are 0 and 1 for fake and real images, respectively), we can use the binary cross-entropy loss function. Therefore, we can determine the ground truth labels for the discriminator loss as follows:

$$\begin{array}{c}\text{Ground truth labels}\\ \text{for the discriminator}\end{array} = \begin{cases} 1: & \text{for real images, i.e.,}\ \boldsymbol{x} \\ 0: & \text{for outputs of}\ G\text{, i.e.,}\ G(\boldsymbol{z}) \end{cases}$$

What about the labels to train the generator? As we want the generator to synthesize realistic images, we want to penalize the generator when its outputs are not classified as real by the discriminator. This means that we will assume the ground truth labels for the outputs of the generator to be 1 when computing the loss function for the generator.

Putting all of this together, the following figure displays the individual steps in a simple GAN model:



In the following section, we will implement a GAN from scratch to generate new handwritten digits.

Implementing a GAN from scratch

In this section, we will cover how to implement and train a GAN model to generate new images such as MNIST digits. Since the training on a normal central processing unit (CPU) may take a long time, in the following subsection, we will cover how to set up the Google Colab environment, which will allow us to run the computations on graphics processing units (GPUs).

Training GAN models on Google Colab

Some of the code examples in this chapter may require extensive computational resources that go beyond a commercial laptop or a workstation without a GPU. If you already have an NVIDIA GPU-enabled computing machine available, with CUDA and cuDNN libraries installed, you can use that to speed up the computations.

However, since many of us do not have access to high-performance computing resources, we will use the Google Colaboratory environment (often referred to as Google Colab), which is a free cloud computing service (available in most countries).

Google Colab provides Jupyter Notebook instances that run on the cloud; the notebooks can be saved on Google Drive or GitHub. While the platform provides various different computing resources, such as CPUs, GPUs, and even tensor processing units (TPUs), it is important to highlight that the execution time is currently limited to 12 hours. Therefore, any notebook running longer than 12 hours will be interrupted.

The code blocks in this chapter will need a maximum computing time of two to three hours, so this will not be an issue. However, if you decide to use Google Colab for other projects that take longer than 12 hours, be sure to use checkpointing and save intermediate checkpoints.

Jupyter Notebook

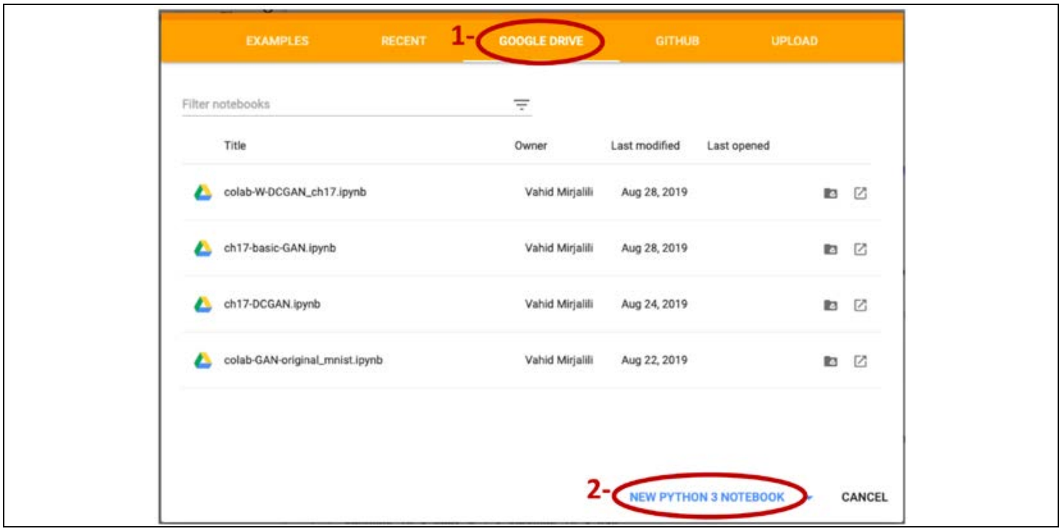
Jupyter Notebook is a graphical user interface (GUI) for running code interactively and interleaving it with text documentation and figures. Due to its versatility and ease of use, it has become one of the most popular tools in data science.

For more information about the general Jupyter Notebook GUI, please view the official documentation at `https://jupyter-notebook.readthedocs.io/en/stable/`. All the code in this book is also available in the form of Jupyter notebooks, and a short introduction can be found in the code directory of the first chapter at `https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch01#pythonjupyter-notebook`.

Lastly, we highly recommend Adam Rule et al.'s article *Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks* on using Jupyter Notebook effectively in scientific research projects, which is freely available at `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007007`.

Accessing Google Colab is very straightforward. You can visit `https://colab.research.google.com`, which automatically takes you to a prompt window where you can see your existing Jupyter notebooks. From this prompt window, click the **GOOGLE DRIVE** tab, as shown in the following figure. This is where you will save the notebook on your Google Drive.

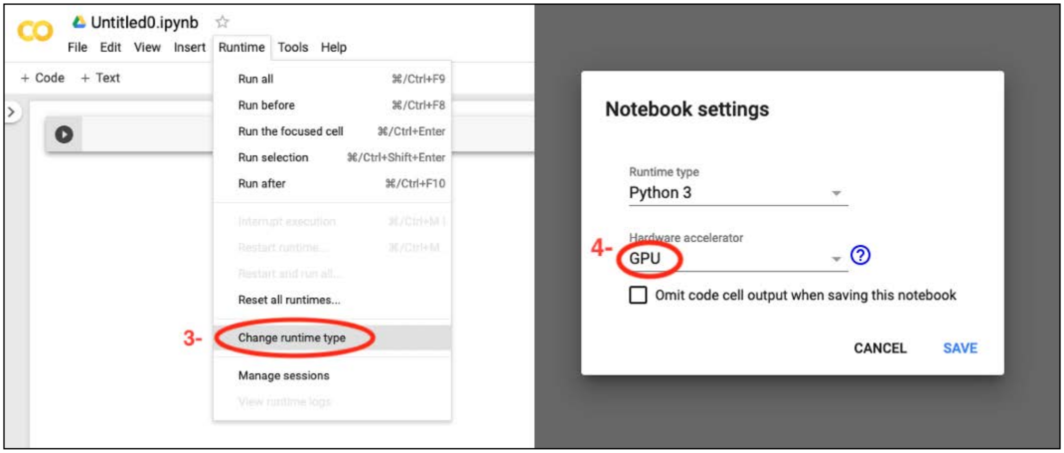
Then, to create a new notebook, click on the link **NEW PYTHON 3 NOTEBOOK** at the bottom of the prompt window:



Title	Owner	Last modified	Last opened
colab-W-DCGAN_ch17.ipynb	Vahid Mirjalili	Aug 28, 2019	
ch17-basic-GAN.ipynb	Vahid Mirjalili	Aug 28, 2019	
ch17-DCGAN.ipynb	Vahid Mirjalili	Aug 24, 2019	
colab-GAN-original_mnist.ipynb	Vahid Mirjalili	Aug 22, 2019	

This will create and open a new notebook for you. All the code examples you write in this notebook will be automatically saved, and you can later access the notebook from your Google Drive in a directory called Colab Notebooks.

In the next step, we want to utilize GPUs to run the code examples in this notebook. To do this, from the **Runtime** option in the menu bar of this notebook, click on **Change runtime type** and select **GPU**, as shown in the following figure:



Notebook settings

Runtime type: Python 3

Hardware accelerator: GPU

□ Omit code cell output when saving this notebook

In the last step, we just need to install the Python packages that we will need for this chapter. The Colab Notebooks environment already comes with certain packages, such as NumPy, SciPy, and the latest stable version of TensorFlow. However, at the time of writing, the latest stable version on Google Colab is TensorFlow 1.15.0, but we want to use TensorFlow 2.0. Therefore, first we need to install TensorFlow 2.0 with GPU support by executing the following command in a new cell of this notebook:

```
! pip install -q tensorflow-gpu==2.0.0
```

(In a Jupyter notebook, a cell starting with an exclamation mark will be interpreted as a Linux shell command.)

Now, we can test the installation and verify that the GPU is available using the following code:

```
>>> import tensorflow as tf
>>> print(tf.__version__)
'2.0.0'
>>> print("GPU Available:", tf.test.is_gpu_available())
GPU Available: True
>>> if tf.test.is_gpu_available():
...     device_name = tf.test.gpu_device_name()
... else:
...     device_name = '/CPU:0'
>>> print(device_name)
'/device:GPU:0'
```

Furthermore, if you want to save the model to your personal Google Drive, or transfer or upload other files, you need to mount the Google Drive. To do this, execute the following in a new cell of the notebook:

```
>>> from google.colab import drive
>>> drive.mount('/content/drive/')
```

This will provide a link to authenticate the Colab Notebook accessing your Google Drive. After following the instructions for authentication, it will provide an authentication code that you need to copy and paste into the designated input field below the cell you have just executed. Then, your Google Drive will be mounted and available at `/content/drive/My Drive`.

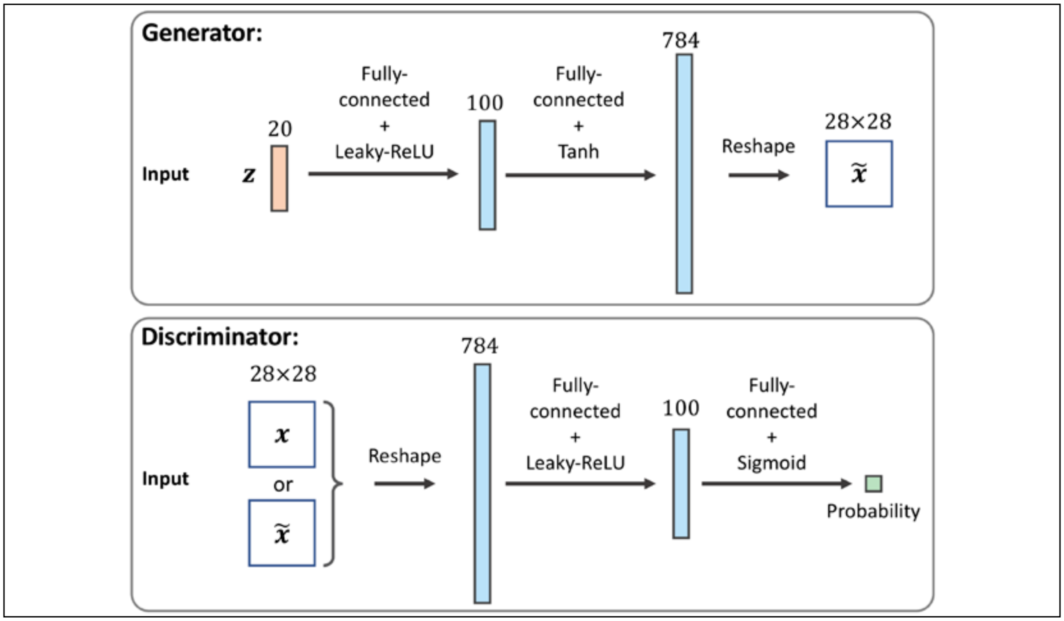
Implementing the generator and the discriminator networks

We will start the implementation of our first GAN model with a generator and a discriminator as two fully connected networks with one or more hidden layers (see the following figure).

This is the original GAN version, which we will refer to as *vanilla GAN*.

In this model, for each hidden layer, we will apply the leaky ReLU activation function. The use of ReLU results in sparse gradients, which may not be suitable when we want to have the gradients for the full range of input values. In the discriminator network, each hidden layer is also followed by a dropout layer. Furthermore, the output layer in the generator uses the hyperbolic tangent (tanh) activation function. (Using tanh activation is recommended for the generator network since it helps with the learning.)

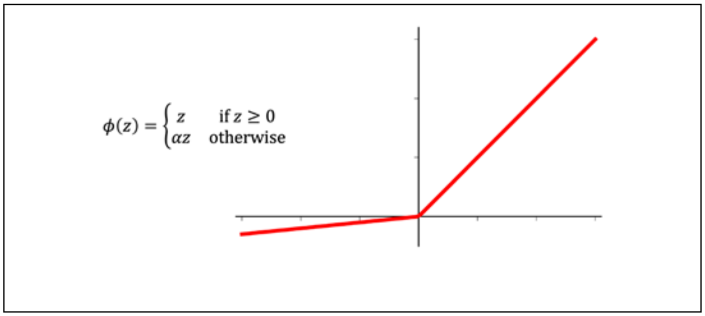
The output layer in the discriminator has no activation function (that is, linear activation) to get the logits. Alternatively, we can use the sigmoid activation function to get probabilities as output:



Leaky rectified linear unit (ReLU) activation function

In *Chapter 13, Parallelizing Neural Network Training with TensorFlow*, we covered different nonlinear activation functions that can be used in an NN model. If you recall, the ReLU activation function was defined as $\phi(z) = \max(0, z)$, which suppresses the negative (preactivation) inputs; that is, negative inputs are set to zero. As a consequence, using the ReLU activation function may result in sparse gradients during backpropagation. Sparse gradients are not always detrimental and can even benefit models for classification. However, in certain applications, such as GANs, it can be beneficial to obtain the gradients for the full range of input values, which we can achieve by making a slight modification to the ReLU function such that it outputs small values for negative inputs. This modified version of the ReLU function is also known as *leaky ReLU*. In short, the leaky ReLU activation function permits non-zero gradients for negative inputs as well, and as a result, it makes the networks more expressive overall.

The leaky ReLU activation function is defined as follows:



$$\phi(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{otherwise} \end{cases}$$

Here, α determines the slope for the negative (preactivation) inputs.

We will define two helper functions for each of the two networks, instantiate a model from the Keras `Sequential` class, and add the layers as described. The code is as follows:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> ## define a function for the generator:
>>> def make_generator_network(
...         num_hidden_layers=1,
...         num_hidden_units=100,
...         num_output_units=784):
...
...     model = tf.keras.Sequential()
...     for i in range(num_hidden_layers):
...         model.add(
...             tf.keras.layers.Dense(
...                 units=num_hidden_units, use_bias=False))
...         model.add(tf.keras.layers.LeakyReLU())
...
...     model.add(
...         tf.keras.layers.Dense(
...             units=num_output_units, activation='tanh'))
...     return model

>>> ## define a function for the discriminator:
>>> def make_discriminator_network(
...         num_hidden_layers=1,
...         num_hidden_units=100,
...         num_output_units=1):
...
...     model = tf.keras.Sequential()
...     for i in range(num_hidden_layers):
...         model.add(
...             tf.keras.layers.Dense(units=num_hidden_units))
...         model.add(tf.keras.layers.LeakyReLU())
...         model.add(tf.keras.layers.Dropout(rate=0.5))
...
...     model.add(
...         tf.keras.layers.Dense(
...             units=num_output_units, activation=None))
...     return model
```

Next, we will specify the training settings for the model. As you will remember from previous chapters, the image size in the MNIST dataset is 28×28 pixels. (That is only one color channel because MNIST contains only grayscale images.) We will further specify the size of the input vector, ***z***, to be 20, and we will use a random uniform distribution to initialize the model weights. Since we are implementing a very simple GAN model for illustration purposes only and using fully connected layers, we will only use a single hidden layer with 100 units in each network. In the following code, we will specify and initialize the two networks, and print their summary information:

```
>>> image_size = (28, 28)
>>> z_size = 20
>>> mode_z = 'uniform' # 'uniform' vs. 'normal'
>>> gen_hidden_layers = 1
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> disc_hidden_size = 100

>>> tf.random.set_seed(1)

>>> gen_model = make_generator_network(
...     num_hidden_layers=gen_hidden_layers,
...     num_hidden_units=gen_hidden_size,
...     num_output_units=np.prod(image_size))

>>> gen_model.build(input_shape=(None, z_size))
>>> gen_model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                multiple                  2000
_________________________________________________________________
leaky_re_lu (LeakyReLU)      multiple                  0
_________________________________________________________________
dense_1 (Dense)              multiple                  79184
=================================================================
Total params: 81,184
Trainable params: 81,184
Non-trainable params: 0
_________________________________________________________________

>>> disc_model = make_discriminator_network(
...     num_hidden_layers=disc_hidden_layers,
...     num_hidden_units=disc_hidden_size)
```

```
>>> disc_model.build(input_shape=(None, np.prod(image_size)))
>>> disc_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_2 (Dense)              multiple                  78500     
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    multiple                  0         
_________________________________________________________________
dropout (Dropout)            multiple                  0         
_________________________________________________________________
dense_3 (Dense)              multiple                  101       
=================================================================
Total params: 78,601
Trainable params: 78,601
Non-trainable params: 0
_________________________________________________________________
```

Defining the training dataset

In the next step, we will load the MNIST dataset and apply the necessary preprocessing steps. Since the output layer of the generator is using the tanh activation function, the pixel values of the synthesized images will be in the range (–1, 1). However, the input pixels of the MNIST images are within the range [0, 255] (with a TensorFlow data type `tf.uint8`). Thus, in the preprocessing steps, we will use the `tf.image.convert_image_dtype` function to convert the `dtype` of the input image tensors from `tf.uint8` to `tf.float32`. As a result, besides changing the `dtype`, calling this function will also change the range of input pixel intensities to [0, 1]. Then, we can scale them by a factor of 2 and shift them by –1 such that the pixel intensities will be rescaled to be in the range [–1, 1]. Furthermore, we will also create a random vector, ***z***, based on the desired random distribution (in this code example, uniform or normal, which are the most common choices), and return both the preprocessed image and the random vector in a tuple:

```
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> mnist = mnist_bldr.as_dataset(shuffle_files=False)

>>> def preprocess(ex, mode='uniform'):
...     image = ex['image']
...     image = tf.image.convert_image_dtype(image, tf.float32)
...     image = tf.reshape(image, [-1])
...     image = image*2 - 1.0
...     if mode == 'uniform':
...         input_z = tf.random.uniform(
...             shape=(z_size,), minval=-1.0, maxval=1.0)
...     elif mode == 'normal':
...         input_z = tf.random.normal(shape=(z_size,))
...     return input_z, image

>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(preprocess)
```

Note that, here, we returned both the input vector, ***z***, and the image to fetch the training data conveniently during model fitting. However, this does not imply that the vector, ***z***, is by any means related to the image—the input image comes from the dataset, while vector ***z*** is generated randomly. In each training iteration, the randomly generated vector, ***z***, represents the input that the generator receives for synthesizing a new image, and the images (the real ones as well as the synthesized ones) are the inputs to the discriminator.

Let's inspect the dataset object that we created. In the following code, we will take one batch of examples and print the array shapes of this sample of input vectors and images. Furthermore, in order to understand the overall data flow of our GAN model, in the following code, we will process a forward pass for our generator and discriminator.

First, we will feed the batch of input, ***z***, vectors to the generator and get its output, `g_output`. This will be a batch of fake examples, which will be fed to the discriminator model to get the logits for the batch of fake examples, `d_logits_fake`. Furthermore, the processed images that we get from the dataset object will be fed to the discriminator model, which will result in the logits for the real examples, `d_logits_real`. The code is as follows:

```
>>> mnist_trainset = mnist_trainset.batch(32, drop_remainder=True)
>>> input_z, input_real = next(iter(mnist_trainset))
>>> print('input-z -- shape:   ', input_z.shape)
>>> print('input-real -- shape:', input_real.shape)
input-z -- shape:    (32, 20)
input-real -- shape: (32, 784)

>>> g_output = gen_model(input_z)
>>> print('Output of G -- shape:', g_output.shape)
Output of G -- shape: (32, 784)

>>> d_logits_real = disc_model(input_real)
>>> d_logits_fake = disc_model(g_output)
>>> print('Disc. (real) -- shape:', d_logits_real.shape)
>>> print('Disc. (fake) -- shape:', d_logits_fake.shape)
Disc. (real) -- shape: (32, 1)
Disc. (fake) -- shape: (32, 1)
```

The two logits, `d_logits_fake` and `d_logits_real`, will be used to compute the loss functions for training the model.

Training the GAN model

As the next step, we will create an instance of `BinaryCrossentropy` as our loss function and use that to calculate the loss for the generator and discriminator associated with the batches that we just processed. To do this, we also need the ground truth labels for each output. For the generator, we will create a vector of 1s with the same shape as the vector containing the predicted logits for the generated images, `d_logits_fake`. For the discriminator loss, we have two terms: the loss for detecting the fake examples involving `d_logits_fake` and the loss for detecting the real examples based on `d_logits_real`.

The ground truth labels for the fake term will be a vector of 0s that we can generate via the `tf.zeros()` (or `tf.zeros_like()`) function. Similarly, we can generate the ground truth values for the real images via the `tf.ones()` (or `tf.ones_like()`) function, which creates a vector of 1s:

```
>>> loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)

>>> ## Loss for the Generator
>>> g_labels_real = tf.ones_like(d_logits_fake)
>>> g_loss = loss_fn(y_true=g_labels_real, y_pred=d_logits_fake)
>>> print('Generator Loss: {:.4f}'.format(g_loss))
Generator Loss: 0.7505

>>> ## Loss for the Discriminator
>>> d_labels_real = tf.ones_like(d_logits_real)
>>> d_labels_fake = tf.zeros_like(d_logits_fake)

>>> d_loss_real = loss_fn(y_true=d_labels_real,
...                       y_pred=d_logits_real)
>>> d_loss_fake = loss_fn(y_true=d_labels_fake,
...                       y_pred=d_logits_fake)
>>> print('Discriminator Losses: Real {:.4f} Fake {:.4f}'
...       .format(d_loss_real.numpy(), d_loss_fake.numpy()))
Discriminator Losses: Real 1.3683 Fake 0.6434
```

The previous code example shows the step-by-step calculation of the different loss terms for the purpose of understanding the overall concept behind training a GAN model. The following code will set up the GAN model and implement the training loop, where we will include these calculations in a `for` loop.

In addition, we will use `tf.GradientTape()` to compute the loss gradients with respect to the model weights and optimize the parameters of the generator and discriminator using two separate Adam optimizers. As you will see in the following code, for alternating between the training of the generator and the discriminator in TensorFlow, we explicitly provide the parameters of each network and apply the gradients of each network separately to the respective designated optimizer:

```
>>> import time
>>> num_epochs = 100
>>> batch_size = 64
>>> image_size = (28, 28)
>>> z_size = 20
>>> mode_z = 'uniform'
>>> gen_hidden_layers = 1
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> disc_hidden_size = 100

>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> if mode_z == 'uniform':
...     fixed_z = tf.random.uniform(
...         shape=(batch_size, z_size),
...         minval=-1, maxval=1)
>>> elif mode_z == 'normal':
...     fixed_z = tf.random.normal(
...         shape=(batch_size, z_size))

>>> def create_samples(g_model, input_z):
...     g_output = g_model(input_z, training=False)
...     images = tf.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0

>>> ## Set-up the dataset
>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(
...     lambda ex: preprocess(ex, mode=mode_z))
```

```
>>> mnist_trainset = mnist_trainset.shuffle(10000)
>>> mnist_trainset = mnist_trainset.batch(
...     batch_size, drop_remainder=True)

>>> ## Set-up the model
>>> with tf.device(device_name):
...     gen_model = make_generator_network(
...         num_hidden_layers=gen_hidden_layers,
...         num_hidden_units=gen_hidden_size,
...         num_output_units=np.prod(image_size))
...     gen_model.build(input_shape=(None, z_size))
...
...     disc_model = make_discriminator_network(
...         num_hidden_layers=disc_hidden_layers,
...         num_hidden_units=disc_hidden_size)
...     disc_model.build(input_shape=(None, np.prod(image_size)))

>>> ## Loss function and optimizers:
>>> loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
>>> g_optimizer = tf.keras.optimizers.Adam()
>>> d_optimizer = tf.keras.optimizers.Adam()

>>> all_losses = []
>>> all_d_vals = []
>>> epoch_samples = []

>>> start_time = time.time()
>>> for epoch in range(1, num_epochs+1):
...
...     epoch_losses, epoch_d_vals = [], []
...
...     for i,(input_z,input_real) in enumerate(mnist_trainset):
...
...         ## Compute generator's loss
...         with tf.GradientTape() as g_tape:
...             g_output = gen_model(input_z)
...             d_logits_fake = disc_model(g_output,
...                                        training=True)
...             labels_real = tf.ones_like(d_logits_fake)
...             g_loss = loss_fn(y_true=labels_real,
...                              y_pred=d_logits_fake)
...
...         ## Compute the gradients of g_loss
...         g_grads = g_tape.gradient(g_loss,
...                       gen_model.trainable_variables)
...
...         ## Optimization: Apply the gradients
...         g_optimizer.apply_gradients(
...             grads_and_vars=zip(g_grads,
...             gen_model.trainable_variables))
...
...         ## Compute discriminator's loss
...         with tf.GradientTape() as d_tape:
...             d_logits_real = disc_model(input_real,
...                                        training=True)
...
...             d_labels_real = tf.ones_like(d_logits_real)
...
...             d_loss_real = loss_fn(
...                 y_true=d_labels_real, y_pred=d_logits_real)
...
...             d_logits_fake = disc_model(g_output,
...                                        training=True)
...             d_labels_fake = tf.zeros_like(d_logits_fake)
...
...             d_loss_fake = loss_fn(
...                 y_true=d_labels_fake, y_pred=d_logits_fake)
...
...             d_loss = d_loss_real + d_loss_fake
...
...         ## Compute the gradients of d_loss
...         d_grads = d_tape.gradient(d_loss,
...                       disc_model.trainable_variables)
...
...         ## Optimization: Apply the gradients
...         d_optimizer.apply_gradients(
...             grads_and_vars=zip(d_grads,
...             disc_model.trainable_variables))
...
...         epoch_losses.append(
...             (g_loss.numpy(), d_loss.numpy(),
...              d_loss_real.numpy(), d_loss_fake.numpy()))
...
...         d_probs_real = tf.reduce_mean(
...                            tf.sigmoid(d_logits_real))
...         d_probs_fake = tf.reduce_mean(
...                            tf.sigmoid(d_logits_fake))
...         epoch_d_vals.append((d_probs_real.numpy(),
...                              d_probs_fake.numpy()))
...
```

```
...     all_losses.append(epoch_losses)
...     all_d_vals.append(epoch_d_vals)
...     print(
...         'Epoch {:03d} | ET {:.2f} min | Avg Losses >>'
...         ' G/D {:.4f}/{:.4f} [D-Real: {:.4f} D-Fake: {:.4f}]'
...         .format(
...             epoch, (time.time() - start_time)/60,
...             *list(np.mean(all_losses[-1], axis=0))))
...     epoch_samples.append(
...         create_samples(gen_model, fixed_z).numpy())
```

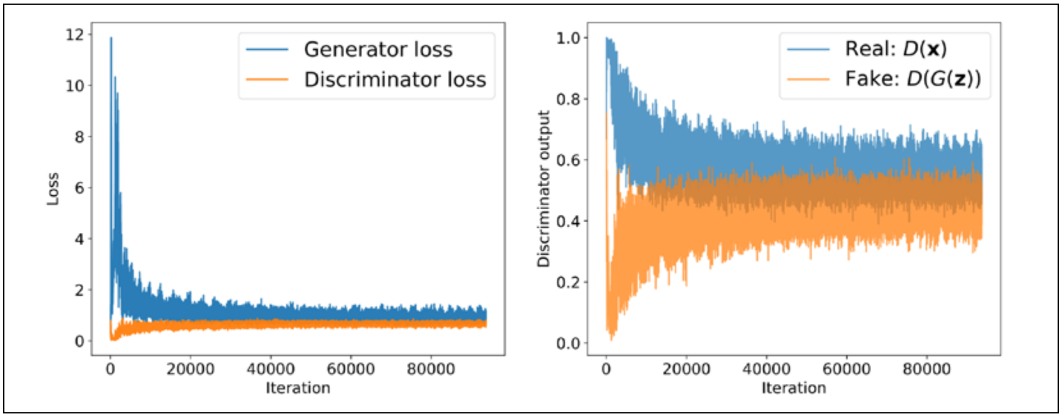
```
Epoch 001 | ET 0.88 min | Avg Losses >> G/D 2.9594/0.2843 [D-Real:
0.0306 D-Fake: 0.2537]

Epoch 002 | ET 1.77 min | Avg Losses >> G/D 5.2096/0.3193 [D-Real:
0.1002 D-Fake: 0.2191]
Epoch ...
Epoch 100 | ET 88.25 min | Avg Losses >> G/D 0.8909/1.3262 [D-Real:
0.6655 D-Fake: 0.6607]
```

Using a GPU, the training process that we implemented in the previous code block should be completed in less than an hour on Google Colab. (It may even be faster on your personal computer if you have a recent and capable CPU and a GPU.) After the model training has completed, it is often helpful to plot the discriminator and generator losses to analyze the behavior of both subnetworks and assess whether they converged.

It is also helpful to plot the average probabilities of the batches of real and fake examples as computed by the discriminator in each iteration. We expect these probabilities to be around 0.5, which means that the discriminator is not able to confidently distinguish between real and fake images:

```
>>> import itertools

>>> fig = plt.figure(figsize=(16, 6))

>>> ## Plotting the losses
>>> ax = fig.add_subplot(1, 2, 1)
>>> g_losses = [item[0] for item in itertools.chain(*all_losses)]
>>> d_losses = [item[1]/2.0 for item in itertools.chain(
...             *all_losses)]
>>> plt.plot(g_losses, label='Generator loss', alpha=0.95)
>>> plt.plot(d_losses, label='Discriminator loss', alpha=0.95)
>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Iteration', size=15)
>>> ax.set_ylabel('Loss', size=15)

>>> epochs = np.arange(1, 101)
>>> epoch2iter = lambda e: e*len(all_losses[-1])
>>> epoch_ticks = [1, 20, 40, 60, 80, 100]
>>> newpos = [epoch2iter(e) for e in epoch_ticks]
>>> ax2 = ax.twiny()
>>> ax2.set_xticks(newpos)
>>> ax2.set_xticklabels(epoch_ticks)
>>> ax2.xaxis.set_ticks_position('bottom')
>>> ax2.xaxis.set_label_position('bottom')
>>> ax2.spines['bottom'].set_position(('outward', 60))
>>> ax2.set_xlabel('Epoch', size=15)
>>> ax2.set_xlim(ax.get_xlim())
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax2.tick_params(axis='both', which='major', labelsize=15)

>>> ## Plotting the outputs of the discriminator
>>> ax = fig.add_subplot(1, 2, 2)
>>> d_vals_real = [item[0] for item in itertools.chain(
...                *all_d_vals)]
>>> d_vals_fake = [item[1] for item in itertools.chain(
...                *all_d_vals)]
>>> plt.plot(d_vals_real, alpha=0.75,
...          label=r'Real: $D(\mathbf{x})$')
>>> plt.plot(d_vals_fake, alpha=0.75,
...          label=r'Fake: $D(G(\mathbf{z}))$')
>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Iteration', size=15)
>>> ax.set_ylabel('Discriminator output', size=15)

>>> ax2 = ax.twiny()
>>> ax2.set_xticks(newpos)
>>> ax2.set_xticklabels(epoch_ticks)
>>> ax2.xaxis.set_ticks_position('bottom')
>>> ax2.xaxis.set_label_position('bottom')
>>> ax2.spines['bottom'].set_position(('outward', 60))
>>> ax2.set_xlabel('Epoch', size=15)
>>> ax2.set_xlim(ax.get_xlim())
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax2.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

The following figure shows the results:



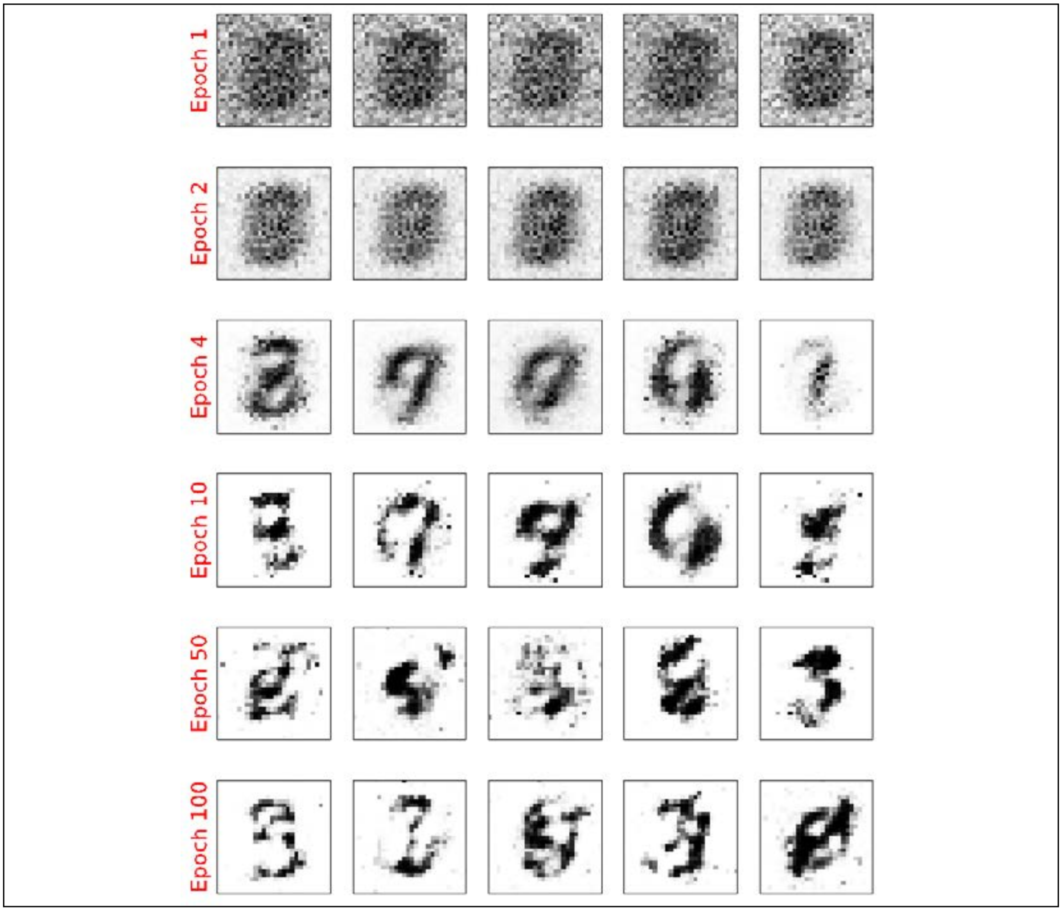
Note that the discriminator model outputs logits, but for this visualization, we already stored the probabilities computed via the sigmoid function before calculating the averages for each batch.

As you can see from the discriminator outputs in the previous figure, during the early stages of the training, the discriminator was able to quickly learn to distinguish quite accurately between the real and fake examples, that is, the fake examples had probabilities close to 0, and the real examples had probabilities close to 1. The reason for that was that the fake examples were nothing like the real ones; therefore, distinguishing between real and fake was rather easy. As the training proceeds further, the generator will become better at synthesizing realistic images, which will result in probabilities of both real and fake examples that are close to 0.5.

Furthermore, we can also see how the outputs of the generator, that is, the synthesized images, change during training. After each epoch, we generated some examples by calling the `create_samples()` function and stored them in a Python list. In the following code, we will visualize some of the images produced by the generator for a selection of epochs:

```
>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         if j == 0:
...             ax.text(
...                 -0.06, 0.5, 'Epoch {}'.format(e),
...                 rotation=90, size=18, color='red',
...                 horizontalalignment='right',
...                 verticalalignment='center',
...                 transform=ax.transAxes)
...
...         image = epoch_samples[e-1][j]
...         ax.imshow(image, cmap='gray_r')
...
>>> plt.show()
```

The following figure shows the produced images:



As you can see from the previous figure, the generator network produced more and more realistic images as the training progressed. However, even after 100 epochs, the produced images still look very different to the handwritten digits contained in the MNIST dataset.

In this section, we designed a very simple GAN model with only a single fully connected hidden layer for both the generator and discriminator. After training the GAN model on the MNIST dataset, we were able to achieve promising, although not yet satisfactory, results with the new handwritten digits. As we learned in *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*, NN architectures with convolutional layers have several advantages over fully connected layers when it comes to image classification. In a similar sense, adding convolutional layers to our GAN model to work with image data might improve the outcome. In the next section, we will implement a **deep convolutional GAN** (**DCGAN**), which uses convolutional layers for both the generator and the discriminator networks.

Improving the quality of synthesized images using a convolutional and Wasserstein GAN

In this section, we will implement a DCGAN, which will enable us to improve the performance we saw in the previous GAN example. Additionally, we will employ several extra key techniques and implement a **Wasserstein GAN** (**WGAN**).

The techniques that we will cover in this section will include the following:

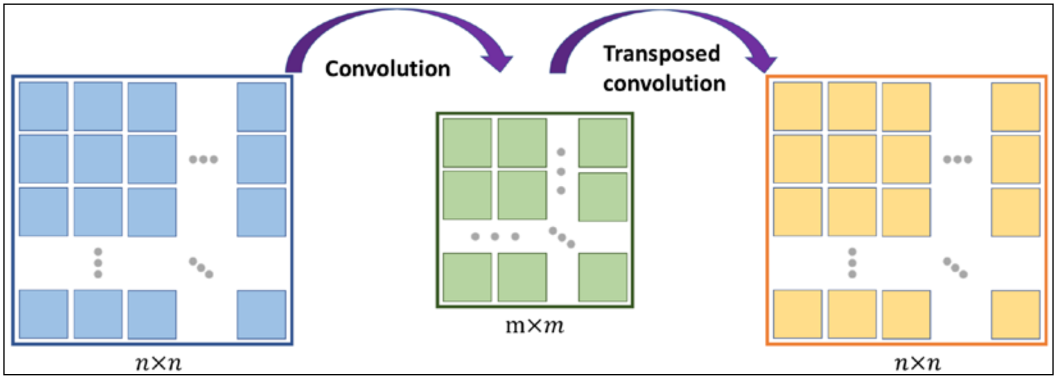
- Transposed convolution
- BatchNorm
- WGAN
- Gradient penalty

The DCGAN was proposed in 2016 by A. Radford, L. Metz, and S. Chintala in their article *Unsupervised representation learning with deep convolutional generative adversarial networks*, which is freely available at `https://arxiv.org/pdf/1511.06434.pdf`. In this article, the researchers proposed using convolutional layers for both the generator and discriminator networks. Starting from a random vector, $\boldsymbol{z}$, the DCGAN first uses a fully connected layer to project $\boldsymbol{z}$ into a new vector with a proper size so that it can be reshaped into a spatial convolution representation ($h \times w \times c$), which is smaller than the output image size. Then, a series of convolutional layers, known as **transposed convolution**, are used to upsample the feature maps to the desired output image size.

Transposed convolution

In *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*, you learned about the convolution operation in one- and two-dimensional spaces. In particular, we looked at how the choices for the padding and strides change the output feature maps. While a convolution operation is usually used to downsample the feature space (for example, by setting the stride to 2, or by adding a pooling layer after a convolutional layer), a *transposed convolution* operation is usually used for *upsampling* the feature space.

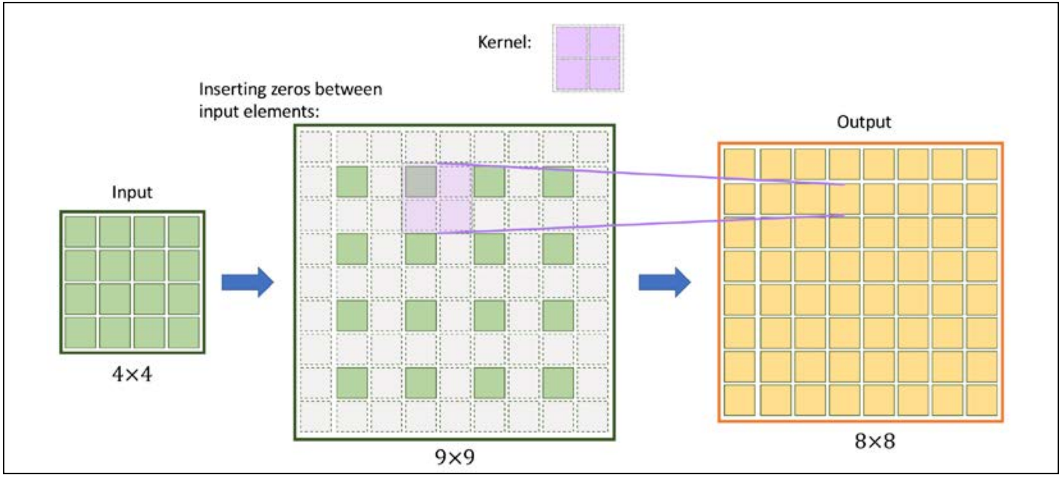
To understand the transposed convolution operation, let's go through a simple thought experiment. Assume that we have an input feature map of size $n \times n$. Then, we apply a 2D convolution operation with certain padding and stride parameters to this $n \times n$ input, resulting in an output feature map of size $m \times m$. Now, the question is, how we can apply another convolution operation to obtain a feature map with the initial dimension $n \times n$ from this $m \times m$ output feature map while maintaining the connectivity patterns between the input and output? Note that only the shape of the $n \times n$ input matrix is recovered and not the actual matrix values. This is what transposed convolution does, as shown in the following figure:



Transposed convolution versus deconvolution

Transposed convolution is also called *fractionally strided convolution*. In deep learning literature, another common term that is used to refer to transposed convolution is *deconvolution*. However, note that deconvolution was originally defined as the inverse of a convolution operation, f, on a feature map, $\boldsymbol{x}$, with weight parameters, $\boldsymbol{w}$, producing feature map $\boldsymbol{x}'$, $f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{x}'$. A deconvolution function, f^{-1}, can then be defined as $f_{\boldsymbol{w}}^{-1}\big(f(\boldsymbol{x})\big) = \boldsymbol{x}$. However, note that the transposed convolution is merely focused on recovering the dimensionality of the feature space and not the actual values.

Upsampling feature maps using transposed convolution works by inserting 0s between the elements of the input feature maps. The following illustration shows an example of applying transposed convolution to an input of size 4 × 4, with a stride of 2 × 2 and kernel size of 2 × 2. The matrix of size 9 × 9 in the center shows the results after inserting such 0s into the input feature map. Then, performing a normal convolution using the 2 × 2 kernel with a stride of 1 results in an output of size 8 × 8. We can verify the backward direction by performing a regular convolution on the output with a stride of 2, which results in an output feature map of size 4 × 4, which is the same as the original input size:



The preceding illustration shows how transposed convolution works in general. There are various cases in which input size, kernel size, strides, and padding variations can change the output. If you want to learn more about all these different cases, refer to the tutorial *A Guide to Convolution Arithmetic for Deep Learning* by Vincent Dumoulin and Francesco Visin, which is freely available at `https://arxiv.org/pdf/1603.07285.pdf`.

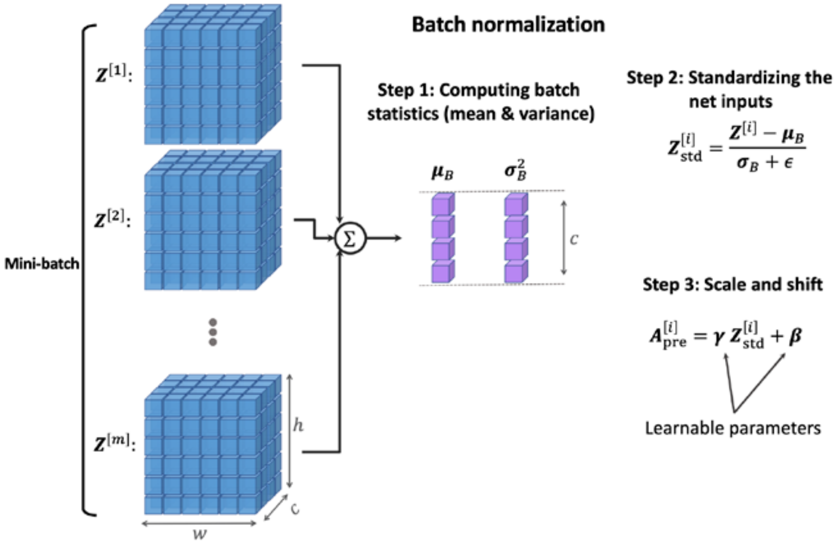
Batch normalization

BatchNorm was introduced in 2015 by Sergey Ioffe and Christian Szegedy in the article *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, which you can access via arXiv at `https://arxiv.org/pdf/1502.03167.pdf`. One of the main ideas behind BatchNorm is normalizing the layer inputs and preventing changes in their distribution during training, which enables faster and better convergence.

BatchNorm transforms a mini-batch of features based on its computed statistics. Assume that we have the net preactivation feature maps obtained after a convolutional layer in a four-dimensional tensor, $\boldsymbol{Z}$, with the shape $[m \times h \times w \times c]$, where m is the number of examples in the batch (i.e., batch size), $h \times w$ is the spatial dimension of the feature maps, and c is the number of channels. BatchNorm can be summarized in three steps, as follows:

1. Compute the mean and standard deviation of the net inputs for each mini-batch:
$$\boldsymbol{\mu}_B = \frac{1}{m \times h \times w} \sum_{i,j,k} \boldsymbol{Z}^{[i,j,k,.]}, \; \boldsymbol{\sigma}_B^2 = \frac{1}{m \times h \times w} \sum_{i,j,k} \left(\boldsymbol{Z}^{[i,j,k,.]} - \boldsymbol{\mu}_B\right)^2$$, where $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ both have size c.
2. Standardize the net inputs for all examples in the batch: $\boldsymbol{Z}_{\text{std}}^{[i]} = \frac{\boldsymbol{Z}^{[i]} - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B + \epsilon}$, where ϵ is a small number for numerical stability (that is, to avoid division by zero).
3. Scale and shift the normalized net inputs using two learnable parameter vectors, $\boldsymbol{\Upsilon}$ and $\boldsymbol{\beta}$, of size c (number of channels): $\boldsymbol{A}_{\text{pre}}^{[i]} = \boldsymbol{\gamma} \boldsymbol{Z}_{\text{std}}^{[i]} + \boldsymbol{\beta}$.

The following figure illustrates the process:


Batch normalization
Mini-batch
$\boldsymbol{Z}^{[1]}$:
$\boldsymbol{Z}^{[2]}$:
$\boldsymbol{Z}^{[m]}$:
h
w
c
Σ
Step 1: Computing batch statistics (mean & variance)
$\boldsymbol{\mu}_B$
$\boldsymbol{\sigma}_B^2$
c
Step 2: Standardizing the net inputs
$\boldsymbol{Z}_{\text{std}}^{[i]} = \frac{\boldsymbol{Z}^{[i]} - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B + \epsilon}$
Step 3: Scale and shift
$\boldsymbol{A}_{\text{pre}}^{[i]} = \boldsymbol{\gamma}\, \boldsymbol{Z}_{\text{std}}^{[i]} + \boldsymbol{\beta}$
Learnable parameters

In the first step of BatchNorm, the mean, $\boldsymbol{\mu}_B$, and standard deviation, $\boldsymbol{\sigma}_B$, of the mini-batch are computed. Both $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ are vectors of size c (where c is the number of channels). Then, these statistics are used in step 2 to scale the examples in each mini-batch via z-score normalization (standardization), resulting in standardized net inputs, $\boldsymbol{Z}_{\text{std}}^{[i]}$. As a consequence, these net inputs are mean-centered and have *unit variance*, which is generally a useful property for gradient descent-based optimization. On the other hand, always normalizing the net inputs such that they have the same properties across the different mini-batches, which can be diverse, can severely impact the representational capacity of NNs. This can be understood by considering a feature, $x \sim N(0,1)$, which, after sigmoid activation to $\sigma(x)$, results in a linear region for values close to 0. Therefore, in step 3, the learnable parameters, $\boldsymbol{\beta}$ and $\boldsymbol{\Upsilon}$, which are vectors of size c (number of channels), allow BatchNorm to control the shift and spread of the normalized features.

During training, the running averages, $\boldsymbol{\mu}_B$, and running variance, $\boldsymbol{\sigma}_B^2$, are computed, which are used along with the tuned parameters, $\boldsymbol{\beta}$ and $\boldsymbol{\Upsilon}$, to normalize the test example(s) at evaluation.

Why does BatchNorm help optimization?

Initially, BatchNorm was developed to reduce the so-called *internal covariance shift*, which is defined as the changes that occur in the distribution of a layer's activations due to the updated network parameters during training.

To explain this with a simple example, consider a fixed batch that passes through the network at epoch 1. We record the activations of each layer for this batch. After iterating through the whole training dataset and updating the model parameters, we start the second epoch, where the previously fixed batch passes through the network. Then, we compare the layer activations from the first and second epochs. Since the network parameters have changed, we observe that the activations have also changed. This phenomenon is called the internal covariance shift, which was believed to decelerate NN training.

However, in 2018, S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry further investigated what makes BatchNorm so effective. In their study, the researchers observed that the effect of BatchNorm on the internal covariance shift is marginal. Based on the outcome of their experiments, they hypothesized that the effectiveness of BatchNorm is, instead, based on a smoother surface of the loss function, which makes the non-convex optimization more robust.

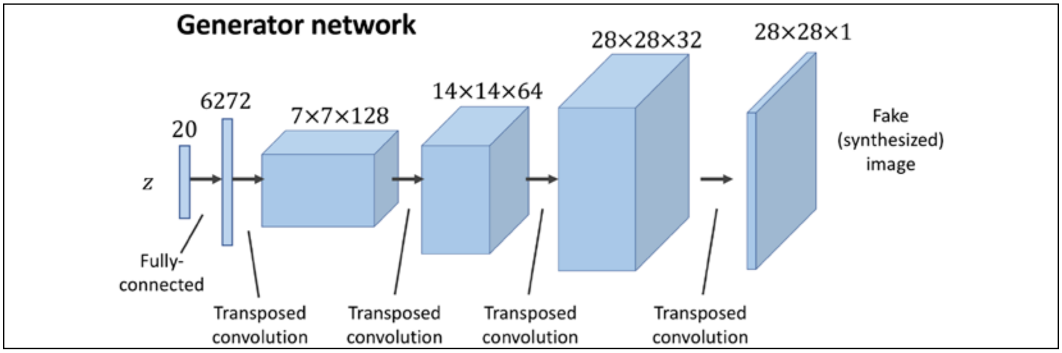
If you are interested in learning more about these results, read through the original paper, *How Does Batch Normalization Help Optimization?*, which is freely available at `http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf`.

The TensorFlow Keras API provides a class, `tf.keras.layers.BatchNormalization()`, that we can use as a layer when defining our models; it will perform all of the steps that we described for BatchNorm. Note that the behavior for updating the learnable parameters, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, depends on whether `training=False` or `training=True`, which can be used to ensure that these parameters are learned only during training.

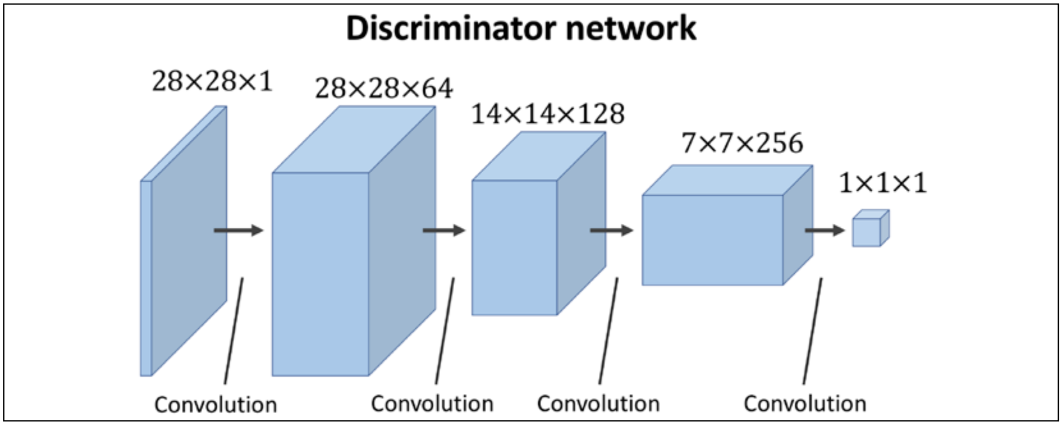
Implementing the generator and discriminator

At this point, we have covered the main components of a DCGAN model, which we will now implement. The architectures of the generator and discriminator networks are summarized in the following two figures.

The generator takes a vector, $\boldsymbol{z}$, of size 20 as input, applies a fully connected (dense) layer to increase its size to 6,272 and then reshapes it into a rank-3 tensor of shape 7 × 7 × 128 (spatial dimension 7 × 7 and 128 channels). Then, a series of transposed convolutions using `tf.keras.layers.Conv2DTransposed()` upsamples the feature maps until the spatial dimension of the resulting feature maps reaches 28 × 28. The number of channels is reduced by half after each transposed convolutional layer, except the last one, which uses only one output filter to generate a grayscale image. Each transposed convolutional layer is followed by BatchNorm and leaky ReLU activation functions, except the last one, which uses tanh activation (without BatchNorm). The architecture for the generator (the feature maps after each layer) is shown in the following figure:



The discriminator receives images of size 28 × 28 × 1, which are passed through four convolutional layers. The first three convolutional layers reduce the spatial dimensionality by 4 while increasing the number of channels of the feature maps. Each convolutional layer is also followed by BatchNorm, leaky ReLU activation, and a dropout layer with `rate=0.3` (drop probability). The last convolutional layer uses kernels of size 7 × 7 and a single filter to reduce the spatial dimensionality of the output to 1 × 1 × 1:



Architecture design considerations for convolutional GANs

Notice that the number of feature maps follows different trends between the generator and the discriminator. In the generator, we start with a large number of feature maps and decrease them as we progress toward the last layer. On the other hand, in the discriminator, we start with a small number of channels and increase it toward the last layer. This is an important point for designing CNNs with the number of feature maps and the spatial size of the feature maps in reverse order. When the spatial size of the feature maps increases, the number of feature maps decreases and vice versa.

In addition, note that it's usually not recommended to use bias units in the layer that follows a BatchNorm layer. Using bias units would be redundant in this case, since BatchNorm already has a shift parameter, $\boldsymbol{\beta}$. You can omit the bias units for a given layer by setting `use_bias=False` in `tf.keras.layers.Dense` or `tf.keras.layers.Conv2D`.

The code for two helper functions to make the generator and discriminator networks is as follows:

```
>>> def make_dcgan_generator(
...         z_size=20,
...         output_size=(28, 28, 1),
...         n_filters=128,
...         n_blocks=2):
...     size_factor = 2**n_blocks
...     hidden_size = (
...         output_size[0]//size_factor,
...         output_size[1]//size_factor)
...
...     model = tf.keras.Sequential([
...         tf.keras.layers.Input(shape=(z_size,)),
...
...         tf.keras.layers.Dense(
...             units=n_filters*np.prod(hidden_size),
...             use_bias=False),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU(),
...         tf.keras.layers.Reshape(
...             (hidden_size[0], hidden_size[1], n_filters)),
...
...         tf.keras.layers.Conv2DTranspose(
...             filters=n_filters, kernel_size=(5, 5),
...             strides=(1, 1), padding='same', use_bias=False),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU()
...     ])
...
...     nf = n_filters
...     for i in range(n_blocks):
...         nf = nf // 2
...         model.add(
...             tf.keras.layers.Conv2DTranspose(
...                 filters=nf, kernel_size=(5, 5),
...                 strides=(2, 2), padding='same',
...                 use_bias=False))
...         model.add(tf.keras.layers.BatchNormalization())
...         model.add(tf.keras.layers.LeakyReLU())
...
...     model.add(
...         tf.keras.layers.Conv2DTranspose(
...             filters=output_size[2], kernel_size=(5, 5),
...             strides=(1, 1), padding='same', use_bias=False,
...             activation='tanh'))
...
```

```
...     return model


>>> def make_dcgan_discriminator(
...         input_size=(28, 28, 1),
...         n_filters=64,
...         n_blocks=2):
...     model = tf.keras.Sequential([
...         tf.keras.layers.Input(shape=input_size),
...         tf.keras.layers.Conv2D(
...             filters=n_filters, kernel_size=5,
...             strides=(1, 1), padding='same'),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU()
...     ])
...
...     nf = n_filters
...     for i in range(n_blocks):
...         nf = nf*2
...         model.add(
...             tf.keras.layers.Conv2D(
...                 filters=nf, kernel_size=(5, 5),
...                 strides=(2, 2),padding='same'))
...         model.add(tf.keras.layers.BatchNormalization())
...         model.add(tf.keras.layers.LeakyReLU())
...         model.add(tf.keras.layers.Dropout(0.3))
...
...     model.add(
...         tf.keras.layers.Conv2D(
...                 filters=1, kernel_size=(7, 7),
...                 padding='valid'))
...
...     model.add(tf.keras.layers.Reshape((1,)))
...
...     return model
```

With these two helper functions, you can build a DCGAN model and train it by using the same MNIST dataset object we initialized in the previous section when we implemented the simple, fully connected GAN. Also, we can use the same loss functions and training procedure as before.

We will be making a few additional modifications to the DCGAN model in the remaining sections of this chapter. Note that the `preprocess()` function for transforming the dataset must change to output an image tensor instead of flattening the image to a vector. The following code shows the necessary modifications to build the dataset, as well as creating the new generator and discriminator networks:

```
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> mnist = mnist_bldr.as_dataset(shuffle_files=False)

>>> def preprocess(ex, mode='uniform'):
...     image = ex['image']
...     image = tf.image.convert_image_dtype(image, tf.float32)
...
...     image = image*2 - 1.0
...     if mode == 'uniform':
...         input_z = tf.random.uniform(
...             shape=(z_size,), minval=-1.0, maxval=1.0)
...     elif mode == 'normal':
...         input_z = tf.random.normal(shape=(z_size,))
...     return input_z, image
```

We can create the generator networks using the helper function, `make_dcgan_generator()`, and print its architecture as follows:

```
>>> gen_model = make_dcgan_generator()
>>> gen_model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6272)              125440

_________________________________________________________________
batch_normalization_7 (Batch (None, 6272)              25088

_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 6272)              0

_________________________________________________________________
reshape_2 (Reshape)          (None, 7, 7, 128)         0

_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 7, 7, 128)         409600

_________________________________________________________________
batch_normalization_8 (Batch (None, 7, 7, 128)         512

_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 7, 7, 128)         0

_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 14, 14, 64)        204800

_________________________________________________________________
batch_normalization_9 (Batch (None, 14, 14, 64)        256

_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 14, 14, 64)        0
```

```
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 28, 28, 32)        51200
_________________________________________________________________
batch_normalization_10 (Batc (None, 28, 28, 32)        128
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 28, 28, 32)        0
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 28, 28, 1)         800
=================================================================
Total params: 817,824
Trainable params: 804,832
Non-trainable params: 12,992
_________________________________________________________________
```

Similarly, we can generate the discriminator network and see its architecture:

```
>>> disc_model = make_dcgan_discriminator()
>>> disc_model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 28, 28, 64)        1664
_________________________________________________________________
batch_normalization_11 (Batc (None, 28, 28, 64)        256
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 28, 28, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 14, 14, 128)       204928
_________________________________________________________________
batch_normalization_12 (Batc (None, 14, 14, 128)       512
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 14, 14, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 7, 7, 256)         819456
_________________________________________________________________
batch_normalization_13 (Batc (None, 7, 7, 256)         1024
_________________________________________________________________
leaky_re_lu_13 (LeakyReLU)   (None, 7, 7, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 7, 7, 256)         0
```

```
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 1, 1, 1)           12545
_________________________________________________________________
reshape_3 (Reshape)          (None, 1)                 0
=================================================================
Total params: 1,040,385
Trainable params: 1,039,489
Non-trainable params: 896
_________________________________________________________________
```

Notice that the number of parameters for the BatchNorm layers is indeed four times the number of channels ($4 \times channels$). Remember that the BatchNorm parameters, $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$, represent the (non-trainable parameters) mean and standard deviation for each feature value inferred from a given batch; $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are the trainable BN parameters.

Note that this particular architecture would not perform very well when using cross-entropy as a loss function.

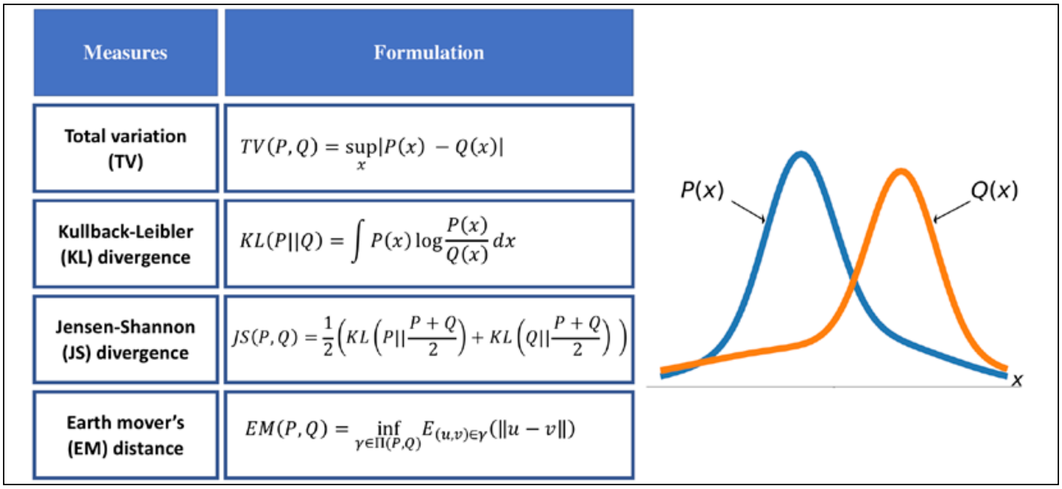
In the next subsection, we will cover WGAN, which uses a modified loss function based on the so-called Wasserstein-1 (or earth mover's) distance between the distributions of real and fake images for improving the training performance.

Dissimilarity measures between two distributions

We will first see different measures for computing the divergence between two distributions. Then, we will see which one of these measures is already embedded in the original GAN model. Finally, switching this measure in GANs will lead us to the implementation of a WGAN.

As mentioned at the beginning of this chapter, the goal of a generative model is to learn how to synthesize new samples that have the same distribution as the distribution of the training dataset. Let $P(x)$ and $Q(x)$ represent the distribution of a random variable, x, as shown in the following figure.

First, let's look at some ways, shown in the following figure, that we can use to measure the dissimilarity between two distributions, *P* and *Q*:

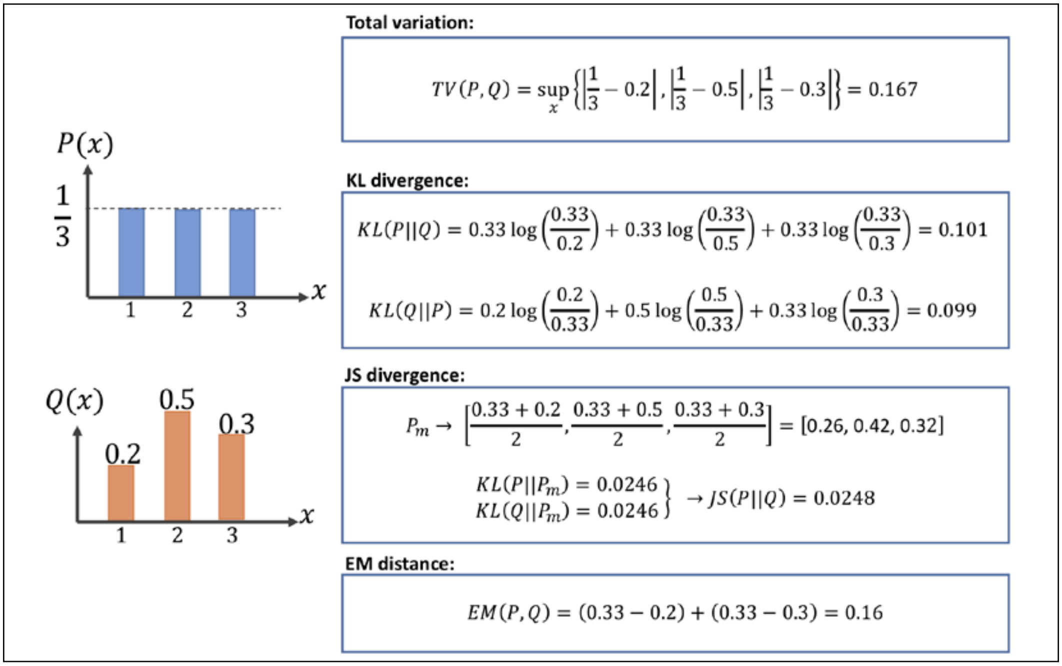


Measures	Formulation
Total variation (TV)	$TV(P,Q) = \sup_{x} \lvert P(x) - Q(x) \rvert$
Kullback-Leibler (KL) divergence	$KL(P\|\|Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Jensen-Shannon (JS) divergence	$JS(P,Q) = \frac{1}{2}\left(KL\left(P\|\|\frac{P+Q}{2}\right) + KL\left(Q\|\|\frac{P+Q}{2}\right)\right)$
Earth mover's (EM) distance	$EM(P,Q) = \inf_{\gamma\in\Pi(P,Q)} E_{(u,v)\in\gamma}(\lVert u - v \rVert)$

The function supremum, sup(*S*), used in the total variation (TV) measure, refers to the smallest value that is greater than all elements of *S*. In other words, sup(*S*) is the least upper bound for *S*. Vice versa, the infimum function, inf(*S*), which is used in EM distance, refers to the largest value that is smaller than all elements of *S* (the greatest lower bound). Let's gain an understanding of these measures by briefly stating what they are trying to accomplish in simple words:

- The first one, TV distance, measures the largest difference between the two distributions at each point.
- The EM distance can be interpreted as the minimal amount of work needed to transform one distribution into the other. The infimum function in the EM distance is taken over $\Pi(\mathrm{P}, \mathrm{Q})$, which is the collection of all joint distributions whose marginals are *P* or *Q*. Then, $\gamma(u, v)$ is a transfer plan, which indicates how we redistribute the earth from location *u* to *v*, subject to some constraints for maintaining valid distributions after such transfers. Computing EM distance is an optimization problem by itself, which is to find the optimal transfer plan, $\gamma(u, v)$.
- The Kullback-Leibler (KL) and Jensen-Shannon (JS) divergence measures come from the field of information theory. Note that KL divergence is not symmetric, that is, $KL(P\|Q) \neq KL(Q\|P)$ in contrast to JS divergence.

The dissimilarity equations provided in the previous figure correspond to continuous distributions but can be extended for discrete cases. An example of calculating these different dissimilarity measures with two simple discrete distributions is illustrated in the following figure:



Total variation:

$$TV(P,Q) = \sup_x \left\{ \left|\frac{1}{3} - 0.2\right|, \left|\frac{1}{3} - 0.5\right|, \left|\frac{1}{3} - 0.3\right| \right\} = 0.167$$

KL divergence:

$$KL(P||Q) = 0.33 \log\left(\frac{0.33}{0.2}\right) + 0.33 \log\left(\frac{0.33}{0.5}\right) + 0.33 \log\left(\frac{0.33}{0.3}\right) = 0.101$$

$$KL(Q||P) = 0.2 \log\left(\frac{0.2}{0.33}\right) + 0.5 \log\left(\frac{0.5}{0.33}\right) + 0.33 \log\left(\frac{0.3}{0.33}\right) = 0.099$$

JS divergence:

$$P_m \rightarrow \left[\frac{0.33+0.2}{2}, \frac{0.33+0.5}{2}, \frac{0.33+0.3}{2}\right] = [0.26, 0.42, 0.32]$$

$$\left.\begin{matrix} KL(P||P_m) = 0.0246 \\ KL(Q||P_m) = 0.0246 \end{matrix}\right\} \rightarrow JS(P||Q) = 0.0248$$

EM distance:

$$EM(P,Q) = (0.33 - 0.2) + (0.33 - 0.3) = 0.16$$

Note that, in the case of the EM distance, for this simple example, we can see that $Q(x)$ at $x = 2$ has the excess value of $0.5 - \frac{1}{3} = 0.166$, while the value of Q at the other two x's is below 1/3. Therefore, the minimal amount of work is when we transfer the extra value at $x = 2$ to $x = 1$ and $x = 3$, as shown in the previous figure. For this simple example, it's easy to see that these transfers will result in the minimal amount of work out of all possible transfers. However, this may be infeasible to do for more complex cases.

The relationship between KL divergence and cross-entropy

KL divergence, $KL(P\|Q)$, measures the relative entropy of the distribution, P, with respect to a reference distribution, Q. The formulation for KL divergence can be extended as

$$KL(P\|Q) = -\int P(x)\log\big(Q(x)\big)\,dx - \left(-\int P(x)\log\big(P(x)\big)\right)$$

Moreover, for discrete distributions, KL divergence can be written as

$$KL(P\|Q) = -\sum_i P(x_i)\frac{P(x_i)}{Q(x_i)}$$

which can be similarly extended as

$$KL(P\|Q) = -\sum_i P(x_i)\log\big(Q(x_i)\big) - \left(-\sum_i P(x_i)\log\big(P(x_i)\big)\right)$$

Based on the extended formulation (either discrete or continuous), KL divergence is viewed as the cross-entropy between P and Q (the first term in the preceding equation) subtracted by the (self-) entropy of P (second term), that is, $KL(P\|Q) = H(P,Q) - H(P)$.

Now, going back to our discussion of GANs, let's see how these different distance measures are related to the loss function for GANs. It can be mathematically shown that the loss function in the original GAN indeed *minimizes the JS divergence between the distribution of real and fake examples*. But, as discussed in an article by Martin Arjovsky et al. (*Wasserstein Generative Adversarial Networks*, `http://proceedings.mlr.press/v70/arjovsky17a/arjovsky17a.pdf`), JS divergence has problems training a GAN model, and therefore, in order to improve the training, the researchers proposed using the EM distance as a measure of dissimilarity between the distribution of real and fake examples.

What is the advantage of using EM distance?

To answer this question, we can consider an example that was given in the article by Martin Arjovsky et al, titled *Wasserstein GAN*. To put it in simple words, assume we have two distributions, P and Q, which are two parallel lines. One line is fixed at $x = 0$ and the other line can move across the x-axis but is initially located at $x = \theta$, where $\theta > 0$.

It can be shown that the KL, TV, and JS dissimilarity measures are $KL(P\|Q) = +\infty$, $TV(P,Q) = 1$, and $JS(P,Q) = \frac{1}{2}\log 2$. None of these dissimilarity measures are a function of the parameter θ, and therefore, they cannot be differentiated with respect to θ toward making the distributions, P and Q, become similar to each other. On the other hand, the EM distance is $EM(P,Q) = |\theta|$, whose gradient with respect to θ exists and can push Q toward P.

Now, let's focus our attention on how EM distance can be used to train a GAN model. Let's assume P_r is the distribution of the real examples and P_g denotes the distributions of fake (generated) examples. P_r and P_g replace P and Q in the EM distance equation. As was mentioned earlier, computing the EM distance is an optimization problem by itself; therefore, this becomes computationally intractable, especially if we want to repeat this computation in each iteration of the GAN training loop. Fortunately, though, the computation of the EM distance can be simplified using a theorem called *Kantorovich-Rubinstein duality*, as follows:

$$W\left(P_r, P_g\right) = \sup_{\|f\|_L \leq 1} E_{u \in P_r}[f(u)] - E_{v \in P_g}[f(v)]$$

Here, the supremum is taken over all the *1-Lipschitz* continuous functions denoted by $\|f\|_L \leq 1$.

Lipschitz continuity

Based on 1-Lipschitz continuity, the function, *f*, must satisfy the following property:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

Furthermore, a real function, $f: R \rightarrow R$, that satisfies the property

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

is called K-Lipschitz continuous.

Using EM distance in practice for GANs

Now, the question is, how do we find such a 1-Lipschitz continuous function to compute the Wasserstein distance between the distribution of real (P_r) and fake (P_g) outputs for a GAN? While the theoretical concepts behind the WGAN approach may seem complicated at first, the answer to this question is simpler than it may appear. Recall that we consider deep NNs to be universal function approximators. This means that we can simply train an NN model to approximate the Wasserstein distance function. As you saw in the previous section, the simple GAN uses a discriminator in the form of a classifier. For WGAN, the discriminator can be changed to behave as a *critic*, which returns a scalar score instead of a probability value. We can interpret this score as how realistic the input images are (like an art critic giving scores to artworks in a gallery).

To train a GAN using the Wasserstein distance, the losses for the discriminator, *D*, and generator, *G*, are defined as follows. The critic (that is, the discriminator network) returns its outputs for the batch of real image examples and the batch of synthesized examples. We use the notations $D(\boldsymbol{x})$ and $D(G(\boldsymbol{z}))$, respectively. Then, the following loss terms can be defined:

- The real component of the discriminator's loss: $L^D_{\text{real}} = -\frac{1}{N}\sum_i D(\boldsymbol{x}_i)$
- The fake component of the discriminator's loss: $L^D_{\text{fake}} = \frac{1}{N}\sum_i D\big(G(\boldsymbol{z}_i)\big)$
- The loss for the generator: $L^G = -\frac{1}{N}\sum_i D\big(G(\boldsymbol{z}_i)\big)$

That will be all for the WGAN, except that we need to ensure that the 1-Lipschitz property of the critic function is preserved during training. For this purpose, the WGAN paper proposes clamping the weights to a small region, for example, [–0.01, 0.01].

Gradient penalty

In the paper by Arjovsky et al., weight clipping is suggested for the 1-Lipschitz property of the discriminator (or critic). However, in another paper titled *Improved Training of Wasserstein GANs*, which is freely available at `https://arxiv.org/pdf/1704.00028.pdf`, Ishaan Gulrajani et al. showed that clipping the weights can lead to exploding and vanishing gradients. Furthermore, weight clipping can also lead to capacity underuse, which means that the critic network is limited to learning only some simple functions, as opposed to more complex functions. Therefore, rather than clipping the weights, Ishaan Guljarani et al. proposed **gradient penalty** (**GP**) as an alternative solution. The result is the **WGAN with gradient penalty** (**WGAN-GP**).

The procedure for the GP that is added in each iteration can be summarized by the following sequence of steps:

1. For each pair of real and fake examples $\left(\boldsymbol{x}^{[i]}, \tilde{\boldsymbol{x}}^{[i]}\right)$ in a given batch, choose a random number, $\alpha^{[i]}$, sampled from a uniform distribution, that is, $\alpha^{[i]} \in U(0,1)$.
2. Calculate an interpolation between the real and fake examples: $\breve{\boldsymbol{x}}^{[i]} = \alpha\boldsymbol{x}^{[i]} + (1-\alpha)\tilde{\boldsymbol{x}}^{[i]}$, resulting in a batch of interpolated examples.
3. Compute the discriminator (critic) output for all the interpolated examples, $D(\breve{\boldsymbol{x}}^{[i]})$.

4. Calculate the gradients of the critic's output with respect to each interpolated example, that is, $\nabla_{\breve{x}^{[i]}} D(\breve{x}^{[i]})$.
5. Compute the GP as $L_{gp}^{D} = \frac{1}{N}\sum_{i}\left(\left\|\nabla_{\breve{x}^{[i]}} D(\breve{x}^{[i]})\right\|_2 - 1\right)^2$.

The total loss for the discriminator is then as follows:

$$L_{\text{total}}^{D} = L_{\text{real}}^{D} + L_{\text{fake}}^{D} + \lambda\, L_{gp}^{D},$$

Here, λ is a tunable hyperparameter.

Implementing WGAN-GP to train the DCGAN model

We have already defined the helper functions that create the generator and discriminator networks for DCGAN (`make_dcgan_generator()` and `make_dcgan_discriminator()`). The code to build the DCGAN model is as follows:

```
>>> num_epochs = 100
>>> batch_size = 128
>>> image_size = (28, 28)
>>> z_size = 20
>>> mode_x = 'uniform'
>>> lambda_gp = 10.0

>>> tf.random.set_seed(1)
>>> np.random.seed(1)


>>> ## Set-up the dataset
>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(preprocess)

>>> mnist_trainset = mnist_trainset.shuffle(10000)
>>> mnist_trainset = mnist_trainset.batch(
...     batch_size, drop_remainder=True)

>>> ## Set-up the model
>>> with tf.device(device_name):
...     gen_model = make_dcgan_generator()
...     gen_model.build(input_shape=(None, z_size))
...
...     disc_model = make_dcgan_discriminator()
...     disc_model.build(input_shape=(None, np.prod(image_size)))
```

Now we can train the model. Note that, typically, the RMSprop optimizer is recommended for WGAN (without the GP), whereas the Adam optimizer is used for WGAN-GP. The code is as follows:

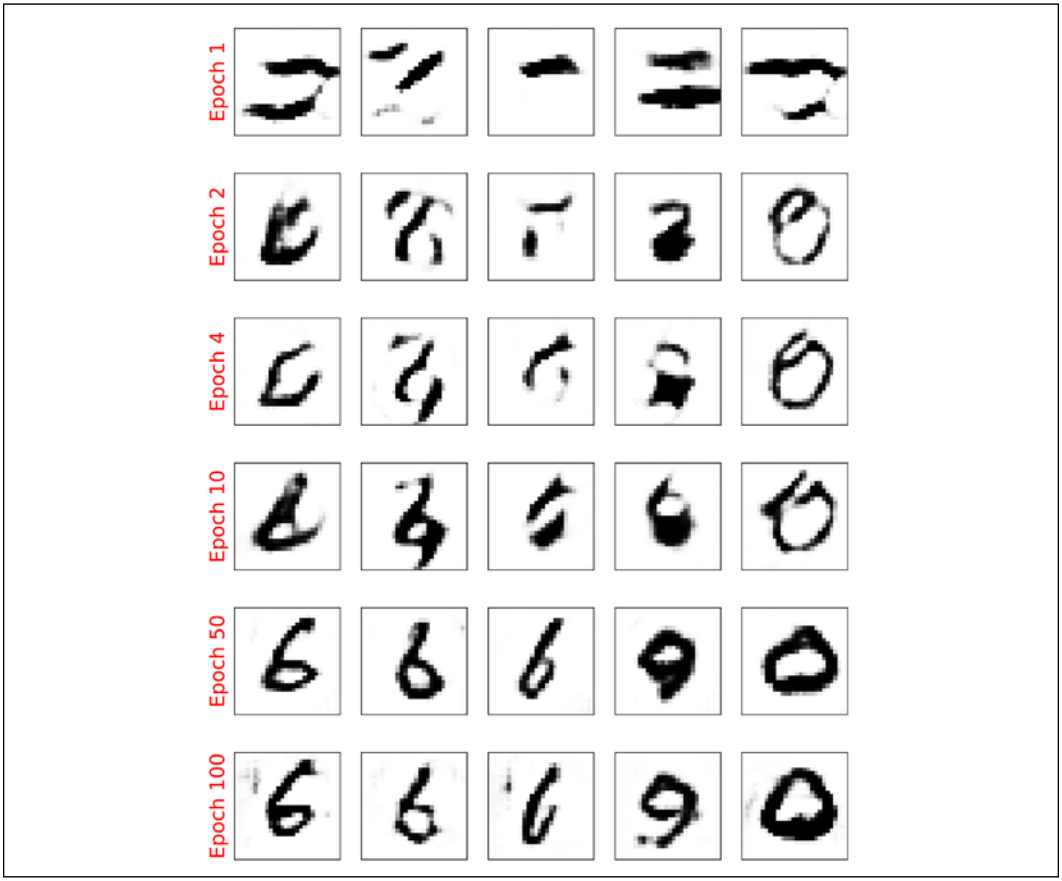
```
>>> import time

>>> ## Optimizers:
>>> g_optimizer = tf.keras.optimizers.Adam(0.0002)
>>> d_optimizer = tf.keras.optimizers.Adam(0.0002)

>>> if mode_z == 'uniform':
...     fixed_z = tf.random.uniform(
...         shape=(batch_size, z_size), minval=-1, maxval=1)
... elif mode_z == 'normal':
...     fixed_z = tf.random.normal(shape=(batch_size, z_size))
...
>>> def create_samples(g_model, input_z):
...     g_output = g_model(input_z, training=False)
...     images = tf.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0

>>> all_losses = []
>>> epoch_samples = []
>>> start_time = time.time()

>>> for epoch in range(1, num_epochs+1):
...
...     epoch_losses = []
...
...     for i,(input_z,input_real) in enumerate(mnist_trainset):
...
...         with tf.GradientTape() as d_tape, tf.GradientTape() \
...                 as g_tape:
...
...             g_output = gen_model(input_z, training=True)
...
...             d_critics_real = disc_model(input_real,
...                 training=True)
...             d_critics_fake = disc_model(g_output,
...                 training=True)
...
...             ## Compute generator's loss:
...             g_loss = -tf.math.reduce_mean(d_critics_fake)
...
```

```
...             ## compute discriminator's losses:
...             d_loss_real = -tf.math.reduce_mean(d_critics_real)
...             d_loss_fake =  tf.math.reduce_mean(d_critics_fake)
...             d_loss = d_loss_real + d_loss_fake
...
...             ## Gradient-penalty:
...             with tf.GradientTape() as gp_tape:
...                 alpha = tf.random.uniform(
...                     shape=[d_critics_real.shape[0], 1, 1, 1],
...                     minval=0.0, maxval=1.0)
...                 interpolated = (alpha*input_real +
...                                  (1-alpha)*g_output)
...                 gp_tape.watch(interpolated)
...                 d_critics_intp = disc_model(interpolated)
...
...             grads_intp = gp_tape.gradient(
...                 d_critics_intp, [interpolated,])[0]
...             grads_intp_l2 = tf.sqrt(
...                 tf.reduce_sum(tf.square(grads_intp),
...                               axis=[1, 2, 3]))
...             grad_penalty = tf.reduce_mean(tf.square(
...                                grads_intp_l2 - 1.0))
...
...             d_loss = d_loss + lambda_gp*grad_penalty
...
...         ## Optimization: Compute the gradients apply them
...         d_grads = d_tape.gradient(d_loss,
...                       disc_model.trainable_variables)
...         d_optimizer.apply_gradients(
...             grads_and_vars=zip(d_grads,
...             disc_model.trainable_variables))
...
...         g_grads = g_tape.gradient(g_loss,
...                       gen_model.trainable_variables)
...         g_optimizer.apply_gradients(
...             grads_and_vars=zip(g_grads,
...             gen_model.trainable_variables))
...
...         epoch_losses.append(
...             (g_loss.numpy(), d_loss.numpy(),
...              d_loss_real.numpy(), d_loss_fake.numpy()))
...
...     all_losses.append(epoch_losses)
...     print(
...         'Epoch {:03d} | ET {:.2f} min | Avg Losses >>'
...         ' G/D {:6.2f}/{:6.2f} [D-Real: {:6.2f}'
...         ' D-Fake: {:6.2f}]'
...         .format(
...             epoch, (time.time() - start_time)/60,
...             *list(np.mean(all_losses[-1], axis=0))))
...     epoch_samples.append(
...         create_samples(gen_model, fixed_z).numpy())
```

Finally, let's visualize the saved examples at some epochs to see how the model is learning and how the quality of synthesized examples changes over the course of learning:

```
>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         if j == 0:
...             ax.text(-0.06, 0.5, 'Epoch {}'.format(e),
...                     rotation=90, size=18, color='red',
...                     horizontalalignment='right',
...                     verticalalignment='center',
...                     transform=ax.transAxes)
...
...         image = epoch_samples[e-1][j]
...         ax.imshow(image, cmap='gray_r')
>>> plt.show()
```

The following figure shows the results:

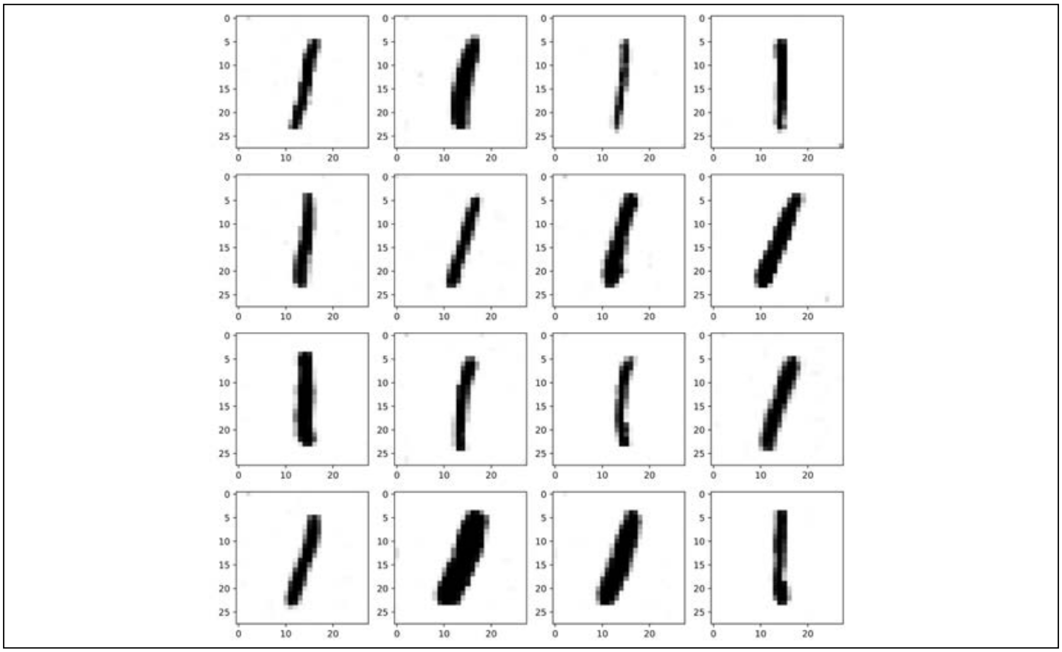


We used the same code to visualize the results as in the section on vanilla GAN. Comparing the new examples shows that DCGAN (with Wasserstein and GP) can generate images of a much higher quality.

Mode collapse

Due to the adversarial nature of GAN models, it is notoriously hard to train them. One common cause of failure in training GANs is when the generator gets stuck in a small subspace and learns to generate similar samples. This is called **mode collapse**, and an example is shown in the following figure.

The synthesized examples in this figure are not cherry-picked. This shows that the generator has failed to learn the entire data distribution, and instead, has taken a lazy approach focusing on a subspace:



Besides the vanishing and exploding gradient problems that we saw previously, there are some further aspects that can also make training GAN models difficult (indeed, it is an art). Here are a few suggested tricks from GAN artists.

One approach is called *mini-batch discrimination*, which is based on the fact that batches consisting of only real or fake examples are fed separately to the discriminator. In mini-batch discrimination, we let the discriminator compare examples across these batches to see whether a batch is real or fake. The diversity of a batch consisting of only real examples is most likely higher than the diversity of a fake batch if a model suffers from mode collapse.

Another technique that is commonly used for stabilizing GAN training is *feature matching*. In feature matching, we make a slight modification to the objective function of the generator by adding an extra term that minimizes the difference between the original and synthesized images based on intermediate representations (feature maps) of the discriminator. We encourage you to read more about this technique in the original article by Ting-Chun Wang et al., titled *High Resolution Image Synthesis and Semantic Manipulation with Conditional GANs,* which is freely available at `https://arxiv.org/pdf/1711.11585.pdf`.

During the training, a GAN model can also get stuck in several modes and just hop between them. To avoid this behavior, you can store some old examples and feed them to the discriminator to prevent the generator from revisiting previous modes. This technique is referred to as *experience replay*. Furthermore, you can train multiple GANs with different random seeds so that the combination of all of them covers a larger part of the data distribution than any single one of them.

Other GAN applications

In this chapter, we mainly focused on generating examples using GANs and looked at a few tricks and techniques to improve the quality of synthesized outputs. The applications of GANs are expanding rapidly, including in computer vision, machine learning, and even other domains of science and engineering. A nice list of different GAN models and application areas can be found at `https://github.com/hindupuravinash/the-gan-zoo`.

It is worth mentioning that we covered GANs in an unsupervised fashion, that is, no class label information was used in the models that were covered in this chapter. However, the GAN approach can be generalized to semi-supervised and supervised tasks, as well. For example, the conditional GAN (cGAN) proposed by Mehdi Mirza and Simon Osindero in the paper *Conditional Generative Adversarial Nets* (`https://arxiv.org/pdf/1411.1784.pdf`) uses the class label information and learns to synthesize new images conditioned on the provided label, that is, $\tilde{\boldsymbol{x}} = G(\boldsymbol{z}|y)$ —applied to MNIST. This allows us to generate different digits in the range 0-9 selectively. Furthermore, conditional GANs allows us to do image-to-image translation, which is to learn how to convert a given image from a specific domain to another. In this context, one interesting work is the Pix2Pix algorithm, published in the paper *Image-to-Image Translation with Conditional Adversarial Networks* by Philip Isola et al. (`https://arxiv.org/pdf/1611.07004.pdf`). It is worth mentioning that in the Pix2Pix algorithm, the discriminator provides the real/fake predictions for multiple patches across the image as opposed to a single prediction for an entire image.

CycleGAN is another interesting GAN model built on top of the cGAN, also for image-to-image translation. However, note that in CycleGAN, the training examples from the two domains are unpaired, meaning that there is no one-to-one correspondence between inputs and outputs. For example, using a CycleGAN, we could change the season of a picture taken in summer to winter. In the paper *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks* by Jun-Yan Zhu et al. (`https://arxiv.org/pdf/1703.10593.pdf`), an impressive example shows horses converted into zebras.

Summary

In this chapter, you first learned about generative models in deep learning and their overall objective: synthesizing new data. We then covered how GAN models use a generator network and a discriminator network, which compete with each other in an adversarial training setting to improve each other. Next, we implemented a simple GAN model using only fully connected layers for both the generator and the discriminator.

We also covered how GAN models can be improved. First, you saw a DCGAN, which uses deep convolutional networks for both the generator and the discriminator. Along the way, you also learned about two new concepts: transposed convolution (for upsampling the spatial dimensionality of feature maps) and BatchNorm (for improving convergence during training).

We then looked at a WGAN, which uses the EM distance to measure the distance between the distributions of real and fake samples. Finally, we talked about the WGAN with GP to maintain the 1-Lipschitz property instead of clipping the weights.

In the next chapter, we will look at reinforcement learning, which is a completely different category of machine learning compared to what we have covered so far in this book.

18

Reinforcement Learning for Decision Making in Complex Environments

In the previous chapters, we focused on supervised and unsupervised machine learning. We also learned how to leverage artificial neural networks and deep learning to tackle problems encountered with these types of machine learning. As you'll recall, supervised learning focuses on predicting a category label or continuous value from a given input feature vector. Unsupervised learning focuses on extracting patterns from data, making it useful for data compression (*Chapter 5*, *Compressing Data via Dimensionality Reduction*), clustering (*Chapter 11*, *Working with Unlabeled Data – Clustering Analysis*), or approximating the training set distribution for generating new data (*Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*).

In this chapter, we turn our attention to a separate category of machine learning, **reinforcement learning** (**RL**), which is different from the previous categories as it is focused on learning *a series of actions* for optimizing an overall reward—for example, winning at a game of chess. In summary, this chapter will cover the following topics:

- Learning the basics of RL, getting familiar with agent/environment interactions, and understanding how the reward process works, in order to help make decisions in complex environments
- Introducing different categories of RL problems, model-based and model-free learning tasks, Monte Carlo, and temporal difference learning algorithms
- Implementing a Q-learning algorithm in a tabular format

- Understanding function approximation for solving RL problems, and combining RL with deep learning by implementing a *deep* Q-learning algorithm

RL is a complex and vast area of research, and this chapter focuses on the fundamentals. As this chapter serves as an introduction, and in order to keep our attention on the important methods and algorithms, we will mainly work with basic examples that illustrate the main concepts. However, toward the end of this chapter, we will go over a more challenging example and utilize deep learning architectures for a particular RL approach, which is known as deep Q-learning.

Introduction – learning from experience

In this section, we will first introduce the concept of RL as a branch of machine learning and see its major differences compared with other tasks of machine learning. After that, we will cover the fundamental components of an RL system. Then, we will see the RL mathematical formulation based on the Markov decision process.

Understanding reinforcement learning

Until this point, this book has primarily focused on *supervised* and *unsupervised* learning. Recall that in *supervised* learning, we rely on labeled training examples, which are provided by a supervisor or a human expert, and the goal is to train a model that can generalize well to unseen, unlabeled test examples. This means that the supervised learning model should learn to assign the same labels or values to a given input example as the supervisor human expert. On the other hand, in *unsupervised* learning, the goal is to learn or capture the underlying structure of a dataset, such as in clustering and dimensionality reduction methods; or learning how to generate new, synthetic training examples with a similar underlying distribution. RL is substantially different from supervised and unsupervised learning, and so RL is often regarded as the "third category of machine learning."

The key element that distinguishes RL from other subtasks of machine learning, such as supervised and unsupervised learning, is that RL is centered around the concept of *learning by interaction*. This means that in RL, the model learns from interactions with an environment to maximize a *reward function*.

While maximizing a reward function is related to the concept of minimizing the cost function in supervised learning, the *correct* labels for learning a series of actions are not known or defined upfront in RL—instead, they need to be learned through interactions with the environment, in order to achieve a certain desired outcome—such as winning at a game. With RL, the model (also called an **agent**) interacts with its environment, and by doing so generates a sequence of interactions that are together called an *episode*. Through these interactions, the agent collects a series of rewards determined by the environment. These rewards can be positive or negative, and sometimes they are not disclosed to the agent until the end of an episode.

For example, imagine that we want to teach a computer to play the game of chess and win against human players. The labels (rewards) for each individual chess move made by the computer are not known until the end of the game, because during the game itself, we don't know whether a particular move will result in winning or losing that game. Only right at the end of the game is the feedback determined. That feedback would likely be a positive reward given if the computer won the game, because the agent had achieved the overall desired outcome; and vice versa, a negative reward would likely be given if the computer had lost the game.

Furthermore, considering the example of playing chess, the input is the current configuration, for instance, the arrangement of the individual chess pieces on the board. Given the large number of possible inputs (the states of the system), it is impossible to label each configuration or state as positive or negative. Therefore, to define a learning process, we provide rewards (or penalties) at the end of each game, when we know whether we reached the desired outcome—whether we won the game or not.

This is the essence of RL. In RL, we cannot or do not teach an agent, computer or robot, *how* to do things; we can only specify *what* we want the agent to achieve. Then, based on the outcome of a particular trial, we can determine rewards depending on the agent's success or failure. This makes RL very attractive for decision making in complex environments—especially when the problem-solving task requires a series of steps, which are unknown, or are hard to explain, or hard to define.

Besides applications in games and robotics, examples of RL can also be found in nature. For example, training a dog involves RL—we hand out rewards (treats) to the dog when it performs certain desirable actions. Or consider a medical dog that is trained to warn its partner of an oncoming seizure. In this case, we do not know the exact mechanism by which the dog is able to detect an oncoming seizure, and we certainly wouldn't be able to define a series of steps to learn seizure detection, even if we had precise knowledge of this mechanism. However, we can reward the dog with a treat if it successfully detects a seizure to *reinforce* this behavior!

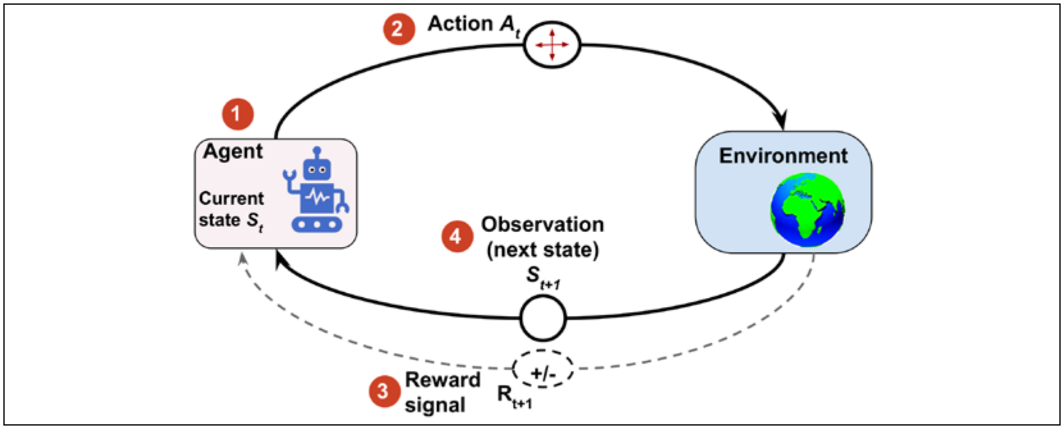
While RL provides a powerful framework for learning an arbitrary series of actions, in order to achieve a certain goal, please do keep in mind that RL is still a relatively young and active area of research with many unresolved challenges. One aspect that makes training RL models particularly challenging is that the consequent model inputs depend on actions taken previously. This can lead to all sorts of problems, and usually results in unstable learning behavior. Also, this sequence-dependence in RL creates a so-called *delayed effect*, which means that the action taken at a time step *t* may result in a future reward appearing some arbitrary number of steps later.

Defining the agent-environment interface of a reinforcement learning system

In all examples of RL, we can find two distinct entities: an agent and an environment. Formally, an **agent** is defined as an entity that learns how to make decisions and interacts with its surrounding environment by taking an action. In return, as a consequence of taking an action, the agent receives observations and a reward signal as governed by the environment. The **environment** is anything that falls outside the agent. The environment communicates with the agent and determines the reward signal for the agent's action as well as its observations.

The **reward signal** is the feedback that the agent receives from interacting with the environment, which is usually provided in the form of a scalar value and can be either positive or negative. The purpose of the reward is to tell the agent how well it has performed. The frequency at which the agent receives the reward depends on the given task or problem. For example, in the game of chess, the reward would be determined after a full game based on the outcome of all the moves: a win or a loss. On the other hand, we could define a maze such that the reward is determined after each time step. In such a maze, the agent then tries to maximize its accumulated rewards over its lifetime—where lifetime describes the duration of an episode.

The following diagram illustrates the interactions and communication between the agent and the environment:



The state of the agent, as illustrated in the previous figure, is the set of all of its variables (1). For example, in the case of a robot drone, these variables could include the drone's current position (longitude, latitude, and altitude), the drone's remaining battery life, the speed of each fan, and so forth. At each time step, the agent interacts with the environment through a set of available actions $\boldsymbol{A_t}$ (2). Based on the action taken by the agent denoted by $\boldsymbol{A_t}$, while it is at state $\boldsymbol{S_t}$, the agent will receive a reward signal $\boldsymbol{R_{t+1}}$ (3), and its state will become $\boldsymbol{S_{t+1}}$ (4).

During the learning process, the agent must try different actions (**exploration**), so that it can progressively learn which actions to prefer and perform more often (**exploitation**) in order to maximize the total, cumulative reward. To understand this concept, let's consider a very simple example where a new computer science graduate with a focus on software engineering is wondering whether to start working at a company (exploitation) or to pursue a Master's or Ph.D. degree to learn more about data science and machine learning (exploration). In general, exploitation will result in choosing actions with a greater short-term reward, whereas exploration can potentially result in greater total rewards in the long run. The tradeoff between exploration and exploitation has been studied extensively, and yet, there is no universal answer to this decision-making dilemma.

The theoretical foundations of RL

Before we jump into some practical examples and start training an RL model, which we will be doing later in this chapter, let's first understand some of the theoretical foundations of RL. The following sections will begin by first examining the mathematical formulation of **Markov decision processes**, episodic versus continuing tasks, some key RL terminology, and dynamic programming using the **Bellman equation**. Let's start with Markov decision processes.

Markov decision processes

In general, the type of problems that RL deals with are typically formulated as **Markov decision processes** (**MDPs**). The standard approach for solving MDP problems is by using dynamic programming, but RL offers some key advantages over dynamic programming.

Dynamic programming

Dynamic programming refers to a set of computer algorithms and programming methods that was developed by Richard Bellman in the 1950s. In a sense, dynamic programming is about recursive problem solving—solving relatively complicated problems by breaking them down into smaller subproblems.

The key difference between recursion and dynamic programming is that dynamic programming stores the results of subproblems (usually as a dictionary or other form of lookup table) so that they can be accessed in constant time (instead of recalculating them) if they are encountered again in future.

Examples of some famous problems in computer science that are solved by dynamic programming include sequence alignment and computing the shortest path from point A to point B.

Dynamic programming is not a feasible approach, however, when the size of states (that is, the number of possible configurations) is relatively large. In such cases, RL is considered a much more efficient and practical alternative approach for solving MDPs.

The mathematical formulation of Markov decision processes

The types of problems that require learning an interactive and sequential decision-making process, where the decision at time step t affects the subsequent situations, are mathematically formalized as Markov decision processes (MDPs).

In case of the agent/environment interactions in RL, if we denote the agent's starting state as S_0, the interactions between the agent and the environment result in a sequence as follows:

$$\{S_0, A_0, R_1\}, \quad \{S_1, A_1, R_2\}, \quad \{S_2, A_2, R_3\}, \quad \ldots$$

Note that the braces serve only as a visual aid. Here, S_t and A_t stand for the state and the action taken at time step t. R_{t+1} denotes the reward received from the environment after performing action A_t. Note that S_t, R_{t+1}, and A_t are time-dependent random variables that take values from predefined finite sets denoted by $s \in \hat{S}$, $r \in \hat{R}$, and $a \in \hat{A}$, respectively. In an MDP, these time-dependent random variables, S_t and R_{t+1}, have probability distributions that only depend on their values at the preceding time step, t – 1. The probability distribution for $S_{t+1} = s'$ and $R_{t+1} = r$ can be written as a conditional probability over the preceding state (S_t) and taken action (A_t) as follows:

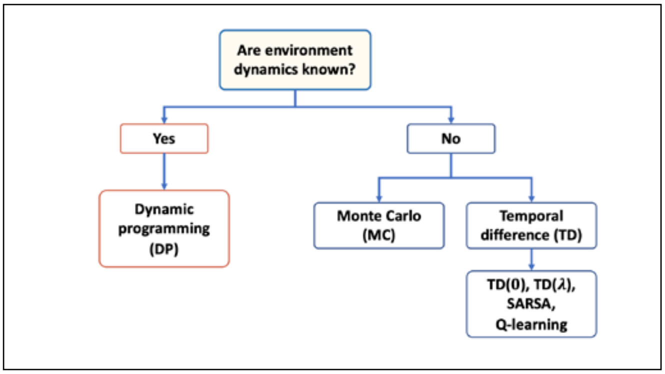
$$p(s', r | s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

This probability distribution completely defines the **dynamics of the environment** (or model of the environment) because, based on this distribution, all transition probabilities of the environment can be computed. Therefore, the environment dynamics are a central criterion for categorizing different RL methods. The types of RL methods that require a model of the environment or try to learn a model of the environment (that is, the environment dynamics) are called *model-based* methods, as opposed to *model-free* methods.

Model-free and model-based RL

When the probability $p(s', r|s, a)$ is known, then the learning task can be solved with dynamic programming. But when the dynamics of the environment are not known, as it is the case in many real-world problems, then you would need to acquire a large number of samples through interacting with the environment to compensate for the unknown environment dynamics.

Two main approaches for dealing with this problem are the model-free Monte Carlo (MC) and temporal difference (TD) methods. The following chart displays the two main categories and the branches of each method:



We will cover these different approaches and their branches from theory to practical algorithms in this chapter.

The environment dynamics can be considered deterministic if particular actions for given states are always or never taken, that is, $p(s', r|s, a) \in \{0,1\}$. Otherwise, in the more general case, the environment would have stochastic behavior.

To make sense of this stochastic behavior, let's consider the probability of observing the future state $S_{t+1} = s'$ conditioned on the current state $S_t = s$ and the performed action $A_t = a$. This is denoted by $p(s'|s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s'|S_t = s, A_t = a)$.

It can be computed as a marginal probability by taking the sum over all possible rewards:

$$p(s'|s,a) \overset{\text{def}}{=} \sum_{r\in\hat{R}} p(s',r|s,a)$$

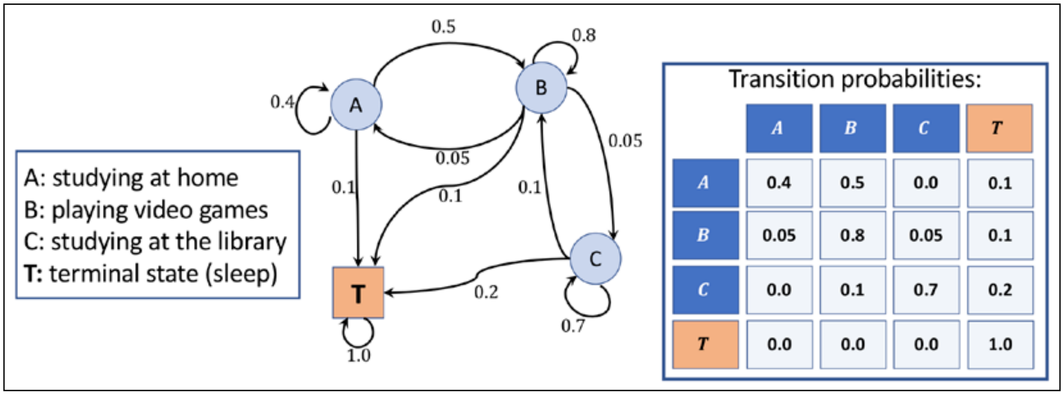
This probability is called **state-transition probability**. Based on the state-transition probability, if the environment dynamics are deterministic, then it means that when the agent takes action $A_t = a$ at state $S_t = s$, the transition to the next state, $S_{t+1} = s'$, will be 100 percent certain, that is, $p(s'|s,a) = 1$.

Visualization of a Markov process

A Markov process can be represented as a directed cyclic graph in which the nodes in the graph represent the different states of the environment. The edges of the graph (that is, the connections between the nodes) represent the transition probabilities between the states.

For example, let's consider a student deciding between three different situations: (A) studying for an exam at home, (B) playing video games at home, or (C) studying at the library. Furthermore, there is a terminal state (T) for going to sleep. The decisions are made every hour, and after making a decision, the student will remain in a chosen situation for that particular hour. Then, assume that when staying at home (state A), there is a 50 percent likelihood that the student switches the activity to playing video games. On the other hand, when the student is at state B (playing video games), there is a relatively high chance (80 percent) that the student will continue playing the video game in the subsequent hours.

The dynamics of the student's behavior is shown as a Markov process in the following figure, which includes a cyclic graph and a transition table:



Transition probabilities:

	A	B	C	T
A	0.4	0.5	0.0	0.1
B	0.05	0.8	0.05	0.1
C	0.0	0.1	0.7	0.2
T	0.0	0.0	0.0	1.0

The values on the edges of the graph represent the transition probabilities of the student's behavior, and their values are also shown in the table to the right. When considering the rows in the table, please note that the transition probabilities coming out of each state (node) always sum to 1.

Episodic versus continuing tasks

As the agent interacts with the environment, the sequence of observations or states forms a trajectory. There are two types of trajectories. If an agent's trajectory can be divided into subparts such that each starts at time *t* = 0 and ends in a terminal state S_T (at *t* = *T*), the task is called an *episodic task*. On the other hand, if the trajectory is infinitely continuous without a terminal state, the task is called a *continuing task*.

The task related to a learning agent for the game of chess is an episodic task, whereas a cleaning robot that is keeping a house tidy is typically performing a continuing task. In this chapter, we only consider episodic tasks.

In episodic tasks, an **episode** is a sequence or trajectory that an agent takes from a starting state, S_0, to a terminal state, S_T:

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_t, A_t, R_{t+1}, \dots, S_{T-1}, A_{T-1}, R_T, S_T$$

For the Markov process shown in the previous figure, which depicts the task of a student studying for an exam, we may encounter episodes like the following three examples:

$$\text{Episode 1: BBCCCCBAT} \rightarrow \text{pass (final reward} = +1)$$
$$\text{Episode 2: ABBBBBBBBBT} \rightarrow \text{fail (final reward} = -1)$$
$$\text{Episode 3: BCCCCCT} \rightarrow \text{pass (final reward} = +1)$$

RL terminology: return, policy, and value function

Next, let's define some additional RL-specific terminology that we will need for the remainder of this chapter.

The return

The so-called return at time *t* is the cumulated reward obtained from the entire duration of an episode. Recall that $R_{t+1} = r$ is the *immediate reward* obtained after performing an action, A_t, at time *t*; the *subsequent* rewards are R_{t+2}, R_{t+3}, and so forth.

The return at time *t* can then be calculated from the immediate reward as well as the subsequent ones, as follows:

$$G_t \overset{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0} \gamma^k R_{t+k+1}$$

Here, γ is the *discount factor* in range [0, 1]. The parameter γ indicates how much the future rewards are "worth" at the current moment (time *t*). Note that by setting $\gamma = 0$, we would imply that we do not care about future rewards. In this case, the return will be equal to the immediate reward, ignoring the subsequent rewards after *t* + 1, and the agent will be short-sighted. On the other hand, if $\gamma = 1$, the return will be the unweighted sum of all subsequent rewards.

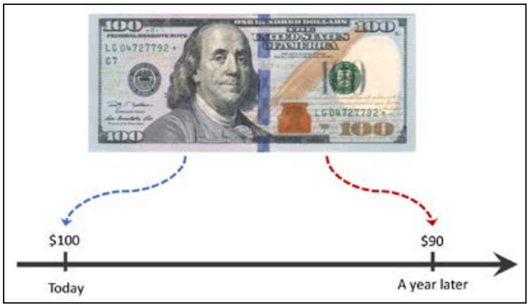
Moreover, note that the equation for the return can be expressed in a simpler way using a *recursion* as follows:

$$G_t = R_{t+1} + \gamma G_{t+1} = r + \gamma G_{t+1}$$

This means that the return at time *t* is equal to the immediate reward *r* plus the discounted future-return at time *t* + 1. This is a very important property, which facilitates the computations of the return.

Intuition behind discount factor

To get an understanding of the discount factor, consider the following figure showing the value of earning a $100 bill today compared to earning it in a year from now. Under certain economic situations, like inflation, earning this $100 bill right now could be worth more than earning it in future:



Therefore, we say that if this bill is worth $100 right now, then it would be worth $90 in a year with a discount factor $\gamma = 0.9$.

Let's compute the return at different time steps for the episodes in our previous student example. Assume $\gamma = 0.9$, and that the only reward given is based on the result of the exam (+1 for passing the exam, and –1 for failing it). The rewards for intermediate time steps are 0.

Episode 1: BBCCCCBAT → pass (final reward = +1):

- $t = 0 \quad : \quad G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^6 R_7$
  $\rightarrow \quad G_0 = 0 + 0 \times \gamma + \cdots + 1 \times \gamma^6 = 0.9^6 \approx 0.531$
- $t = 1 \quad : \quad G_1 = 1 \times \gamma^5 = 0.590$
- $t = 2 \quad : \quad G_2 = 1 \times \gamma^4 = 0.656$
- ...
- $t = 6 \quad : \quad G_6 = 1 \times \gamma = 0.9$
- $t = 7 \quad : \quad G_7 = 1 = 1$

Episode 2: ABBBBBBBBBT → fail (final reward = −1):

- $t = 0 \quad : \quad G_0 = -1 \times \gamma^8 = -0.430$
- $t = 1 \quad : \quad G_0 = -1 \times \gamma^7 = -0.478$
- ...
- $t = 8 \quad : \quad G_0 = -1 \times \gamma = -0.9$
- $t = 9 \quad : \quad G_{10} = -1$

We leave the computation of the returns for the third episode as an exercise for the reader.

Policy

A *policy* typically denoted by $\pi(a|s)$ is a function that determines the next action to take, which can be either deterministic, or stochastic (that is, the probability for taking the next action). A stochastic policy then has a probability distribution over actions that an agent can take at a given state:

$$\pi(a|s) \stackrel{\text{def}}{=} P[A_t = a \,|S_t = s]$$

During the learning process, the policy may change as the agent gains more experience. For example, the agent may start from a random policy, where the probability of all actions is uniform; meanwhile, the agent will hopefully learn to optimize its policy toward reaching the optimal policy. The *optimal policy* $\pi_*(a|s)$ is the policy that yields the highest return.

Value function

The *value function*, also referred to as the *state-value function*, measures the *goodness* of each state—in other words, how good or bad it is to be in a particular state. Note that the criterion for goodness is based on the return.

Now, based on the return G_t, we define the value function of state s as the expected return (the average return over all possible episodes) after *following policy* π:

$$v_\pi(s) \overset{\text{def}}{=} E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0} \gamma^{k+1} R_{t+k+1} \middle| S_t = s\right]$$

In an actual implementation, we usually estimate the value function using lookup tables, so we do not have to recompute it multiple times. (This is the dynamic programming aspect.) For example, in practice, when we estimate the value function using such tabular methods, we store all the state values in a table denoted by $V(s)$. In a Python implementation, this could be a list or a NumPy array whose indexes refer to different states; or, it could be a Python dictionary, where the dictionary keys map the states to the respective values.

Moreover, we can also define a value for each state-action pair, which is called the *action-value function* and is denoted by $q_\pi(s, a)$. The action-value function refers to the expected return G_t when the agent is at state $S_t = s$ and takes action $A_t = a$. Extending the definition of state-value function to state-action pairs, we get the following:

$$q_\pi(s, a) \overset{\text{def}}{=} E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0} \gamma^{k+1} R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Similar to referring to the optimal policy as $\pi_*(a|s)$, $v_*(s)$ and $q_*(s, a)$ also denote the optimal state-value and action-value functions.

Estimating the value function is an essential component of RL methods. We will cover different ways of calculating and estimating the state-value function and action-value function later in this chapter.

The difference between the reward, return, and value function

The **reward** is a consequence of the agent taking an action given the current state of the environment. In other words, the reward is a signal that the agent receives when performing an action to transition from one state to the next. However, remember that not every action yields a positive or negative reward—think back to our chess example where a positive reward is only received upon winning the game, and the reward for all intermediate actions is zero.

A state itself has a certain value, which we assign to it to measure how good or bad this state is – this is where the value function comes into play. Typically, the states with a "high" or "good" value are those states that have a high expected **return** and will likely yield a high reward given a particular policy.

For example, let's consider a chess-playing computer once more. A positive reward may only be given at the end of the game if the computer wins the game. There is no (positive) reward if the computer loses the game. Now, imagine the computer performs a particular chess move that captures the opponent's queen without any negative consequences for the computer. Since the computer only receives a reward for winning the game, it does not get an immediate reward by making this move that captures the opponent's queen. However, the new state (the state of the board after capturing the queen) may have a **high value**, which may yield a reward (if the game is won afterward). Intuitively, we can say that the high value associated with capturing the opponent's queen is associated with the fact that capturing the queen often results in winning the game—and thus the high expected return, or value. However, note that capturing the opponent's queen does not always lead to winning the game; hence, the agent is likely to receive a positive reward, but it is not guaranteed.

In short, the **return** is the weighted sum of **rewards** for an entire episode, which would be equal to the discounted final reward in our chess example (since there is only one reward). The **value function** is the expectation over all possible episodes, which basically computes how "valuable" it is on average to make a certain move.

Before we move directly ahead into some RL algorithms, let's briefly go over the derivation for the Bellman equation, which we can use to implement the policy evaluation.

Dynamic programming using the Bellman equation

The Bellman equation is one of the central elements of many RL algorithms. The Bellman equation simplifies the computation of the value function, such that rather than summing over multiple time steps, it uses a recursion that is similar to the recursion for computing the return.

Based on the recursive equation for the total return $G_t = r + \gamma G_{t+1}$, we can rewrite the value function as follows:

$$\begin{aligned} v_\pi(s) &\overset{\text{def}}{=} E_\pi[G_t | S_t = s] \\ &= E_\pi[r + \gamma G_{t+1} | S_t = s] \\ &= r + \gamma E_\pi[G_{t+1} | S_t = s] \end{aligned}$$

Notice that the immediate reward *r* is taken out of the expectation since it is a constant and known quantity at time *t*.

Similarly, for the action-value function, we could write:

$$\begin{aligned} q_\pi(s,a) &\overset{\text{def}}{=} E_\pi[G_t | S_t = s, A_t = a] \\ &= E_\pi[r + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= r + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

We can use the environment dynamics to compute the expectation via summing over all probabilities of the next state s' and the corresponding rewards *r*:

$$v_\pi(s) = \sum_{a \in \widehat{A}} \pi(a|s) \sum_{s' \in \hat{S}, r' \in \hat{R}} p(s', r | s, a) \left[r + \gamma E_\pi[G_{t+1} | S_{t+1} = s'] \right]$$

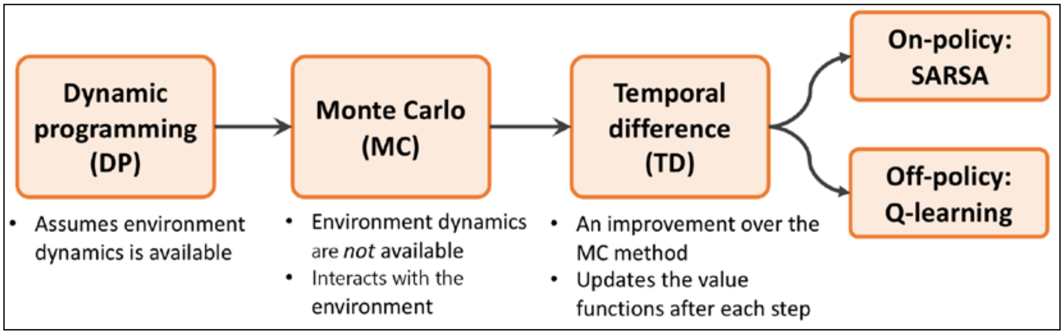
Now, we can see that expectation of the return, $E_\pi[G_{t+1} | S_{t+1} = s']$, is essentially the state-value function $v_\pi(s')$. So, we can write $v_\pi(s)$ as a function of $v_\pi(s')$:

$$v_\pi(s) = \sum_{a \in \hat{A}} \pi(a|s) \sum_{s' \in \hat{S}, r' \in \hat{R}} p(s', r' | s, a) \left[r' + \gamma v_\pi(s') \right]$$

This is called the **Bellman equation**, which relates the value function for a state, s, to the value function of its subsequent state, s'. This greatly simplifies the computation of the value function because it eliminates the iterative loop along the time axis.

Reinforcement learning algorithms

In this section, we will cover a series of learning algorithms. We will start with dynamic programming, which assumes that the transition dynamics (or the environment dynamics, that is, $p(s', r|s, a)$, are known. However, in most RL problems, this is not the case. To work around the unknown environment dynamics, RL techniques were developed that learn through interacting with the environment. These techniques include MC, TD learning, and the increasingly popular Q-learning and deep Q-learning approaches. The following figure describes the course of advancing RL algorithms, from dynamic programming to Q-learning:


Dynamic programming (DP)
Monte Carlo (MC)
Temporal difference (TD)
On-policy: SARSA
Off-policy: Q-learning
• Assumes environment dynamics is available
• Environment dynamics are *not* available
• Interacts with the environment
• An improvement over the MC method
• Updates the value functions after each step

In the following sections of this chapter, we will step through each of these RL algorithms. We will start with dynamic programming, before moving on to MC, and finally on to TD and its branches of on-policy **SARSA** (**state–action–reward–state–action**) and off-policy Q-learning. We will also move into deep Q-learning while we build some practical models.

Dynamic programming

In this section, we will focus on solving RL problems under the following assumptions:

- We have full knowledge about the environment dynamics; that is, all transition probabilities $p(s', r'|s, a)$ are known.

- The agent's state has the Markov property, which means that the next action and reward only depend on the current state and the choice of action we make at this moment or current time step.

The mathematical formulation for RL problems using a Markov decision process (MDP) was introduced earlier in this chapter. If you need a refresher, please refer to the section called *The mathematical formulation of Markov decision processes*, which introduced the formal definition of the value function $v_\pi(s)$ following the policy π, and the Bellman equation, which was derived using the environment dynamics.

We should emphasize that dynamic programming is not a practical approach for solving RL problems. The problem with using dynamic programming is that it assumes full knowledge of the environment dynamics, which is usually unreasonable or impractical for most real-world applications. However, from an educational standpoint, dynamic programming helps with introducing RL in a simple fashion and motivates the use of more advanced and complicated RL algorithms.

There are two main objectives via the tasks described in the following subsections:

1. Obtain the true state-value function, $v_\pi(s)$; this task is also known as the prediction task and is accomplished with *policy evaluation*.
2. Find the optimal value function, $v_*(s)$, which is accomplished via *generalized policy iteration*.

Policy evaluation – predicting the value function with dynamic programming

Based on the Bellman equation, we can compute the value function for an arbitrary policy π with dynamic programming when the environment dynamics are known. For computing this value function, we can adapt an iterative solution, where we start from $v^{\langle 0\rangle}(s)$, which is initialized to zero values for each state. Then, at each iteration $i + 1$, we update the values for each state based on the Bellman equation, which is, in turn, based on the values of states from a previous iteration i, as follows:

$$v^{\langle i+1\rangle}(s) = \sum_{a} \pi(a|s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r|s, a)\left[r + \gamma v^{\langle i\rangle}(s')\right]$$

It can be shown that as the iterations increase to infinity, $v^{\langle i\rangle}(s)$ converges to the true state-value function $v_\pi(s)$.

Also, notice here that we do not need to interact with the environment. The reason for this is that we already know the environment dynamics accurately. As a result, we can leverage this information and estimate the value function easily.

After computing the value function, an obvious question is how that value function can be useful for us if our policy is still a random policy. The answer is that we can actually use this computed $v_\pi(s)$ to improve our policy, as we will see next.

Improving the policy using the estimated value function

Now that we have computed the value function $v_\pi(s)$ by following the existing policy, π, we want to use $v_\pi(s)$ and improve the existing policy, π. This means that we want to find a new policy, π', that for each state, *s*, following π' would yield higher or at least equal value than using the current policy, π. In mathematical terms, we can express this objective for the improved policy, π', as:

$$v_{\pi'}(s) \geq v_\pi(s) \qquad \forall s \in \hat{S}$$

First, recall that a policy, π, determines the probability of choosing each action, *a,* while the agent is at state *s*. Now, in order to find π' that always has a better or equal value for each state, we first compute the action-value function, $q_\pi(s, a)$, for each state, *s,* and action, *a*, based on the computed state value using the value function $v_\pi(s)$. We iterate through all the states, and for each state, *s*, we compare the value of the next state s', that would occur if action *a* was selected.

After we have obtained the highest state value by evaluating all state-action pairs via $q_\pi(s, a)$, we can compare the corresponding action with the action selected by the current policy. If the action suggested by the current policy (that is, $\arg\max_a \pi(a|s)$) is different than the action suggested by the action-value function (that is, $\arg\max_a q_\pi(s, a)$), then we can update the policy by reassigning the probabilities of actions to match the action that gives the highest action value, $q_\pi(s, a)$. This is called the *policy improvement* algorithm.

Policy iteration

Using the policy improvement algorithm described in the previous subsection, it can be shown that the policy improvement will strictly yield a better policy, unless the current policy is already optimal (which means $v_\pi(s) = v_{\pi'}(s) = v_*(s)$ for each $s \in \hat{s}$). Therefore, if we iteratively perform policy evaluation followed by policy improvement, we are guaranteed to find the optimal policy.

Note that this technique is referred to as **generalized policy iteration** (**GPI**), which is common among many RL methods. We will use the GPI in later sections of this chapter for the MC and TD learning methods.

Value iteration

We saw that by repeating the policy evaluation (compute $v_\pi(s)$ and $q_\pi(s,a)$) and policy improvement (finding π' such that $v_{\pi'}(s) \geq v_\pi(s)\ \forall s \in \hat{S}$), we can reach the optimal policy. However, it can be more efficient if we combine the two tasks of policy evaluation and policy improvement into a single step. The following equation updates the value function for iteration $i+1$ (denoted by $v^{\langle i+1 \rangle}$) based on the action that maximizes the weighted sum of the next state value and its immediate reward ($r + \gamma v^{\langle i \rangle}(s')$):

$$v^{\langle i+1 \rangle}(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v^{\langle i \rangle}(s')\right]$$

In this case, the updated value for $v^{\langle i+1 \rangle}(s)$ is maximized by choosing the best action out of all possible actions, whereas in policy evaluation, the updated value was using the weighted sum over all actions.

Notation for tabular estimates of the state value and action value functions

In most RL literature and textbooks, the lowercase v_π and q_π are used to refer to the true state-value and true action-value functions, respectively, as mathematical functions.

Meanwhile, for practical implementations, these value functions are defined as lookup tables. The tabular estimates of these value functions are denoted by $V(S_t = s) \approx v_\pi(s)$ and $Q_\pi(S_t = s, A_t = a) \approx q_\pi(s,a)$. We will also use this notation in this chapter.

Reinforcement learning with Monte Carlo

As we saw in the previous section on dynamic programming, it relies on a simplistic assumption that the environment's dynamics are fully known. Moving away from the dynamic programming approach, we now assume that we do not have any knowledge about the environment dynamics.

That is, we do not know the state-transition probabilities of the environment, and instead, we want the agent to learn through *interacting* with the environment. Using MC methods, the learning process is based on the so-called *simulated experience*.

For MC-based RL, we define an agent class that follows a probabilistic policy, π, and based on this policy, our agent takes an action at each step. This results in a simulated episode.

Earlier, we defined the state-value function, such that the value of a state indicates the expected return from that state. In dynamic programming, this computation relied on the knowledge of the environment dynamics, that is, $p(s', r|s, a)$.

However, from now on, we will develop algorithms that do not require the environment dynamics. MC-based methods solve this problem by generating simulated episodes where an agent interacts with the environment. From these simulated episodes, we will be able to compute the average return for each state visited in that simulated episode.

State-value function estimation using MC

After generating a set of episodes, for each state, s, the set of episodes that all pass through state s is considered for calculating the value of state s. Let's assume that a lookup table is used for obtaining the value corresponding to the value function, $V(S_t = s)$. MC updates for estimating the value function are based on the total return obtained in that episode starting from the first time that state s is visited. This algorithm is called *first-visit Monte Carlo* value prediction.

Action-value function estimation using MC

When the environment dynamics are known, we can easily infer the action-value function from a state-value function by looking one step ahead to find the action that gives the maximum value, as was shown in the *Dynamic programming* section. However, this is not feasible if the environment dynamics are unknown.

To solve this issue, we can extend the algorithm for estimating the first-visit MC state-value prediction. For instance, we can compute the *estimated* return for each state-action pair using the action-value function. To obtain this estimated return, we consider visits to each state-action pair (s, a), which refers to visiting state s and taking action a.

However, a problem arises since some actions may never be selected, resulting in insufficient exploration. There are a few ways to resolve this. The simplest approach is called *exploratory start*, which assumes that every state-action pair has a non zero probability at the beginning of the episode.

Another approach to deal with this lack-of-exploration issue is called *ϵ-greedy policy*, which will be discussed in the next section on policy improvement.

Finding an optimal policy using MC control

MC control refers to the optimization procedure for improving a policy. Similar to the policy iteration approach in previous section (*Dynamic programming*), we can repeatedly alternate between policy evaluation and policy improvement until we reach the optimal policy. So, starting from a random policy, π_0, the process of alternating between policy evaluation and policy improvement can be illustrated as follows:

$$\pi_0 \xrightarrow{Eval.} q_{\pi_0} \xrightarrow{Improve} \quad \pi_1 \xrightarrow{Eval.} q_{\pi_1} \xrightarrow{Improve} \quad \pi_2 \quad \ldots \quad \xrightarrow{Eval.} q_* \xrightarrow{Improve} \pi_*$$

Policy improvement – computing the greedy policy from the action-value function

Given an action-value function, $q(s, a)$, we can generate a greedy (deterministic) policy as follows:

$$\pi(s) \stackrel{\text{def}}{=} \arg\max_a q(s,a)$$

In order to avoid the lack-of-exploration problem, and to consider the non-visited state-action pairs as discussed earlier, we can let the non-optimal actions have a small chance (ϵ) to be chosen. This is called the ϵ-greedy policy, according to which, all non-optimal actions at state s have a minimal $\frac{\epsilon}{|A(s)|}$ probability of being selected (instead of 0), and the optimal action has a probability of $1 - \frac{(|A(s)| - 1) \times \epsilon}{|A(s)|}$ (instead of 1).

Temporal difference learning

So far, we have seen two fundamental RL techniques, dynamic programming and MC-based learning. Recall that dynamic programming relies on the complete and accurate knowledge of the environment dynamics. The MC-based method, on the other hand, learns by simulated experience. In this section, we will now introduce a third RL method called TD learning, which can be considered as an improvement or extension of the MC-based RL approach.

Similar to the MC technique, TD learning is also based on learning by experience and, therefore, does not require any knowledge of environment dynamics and transition probabilities. The main difference between the TD and MC techniques is that in MC, we have to wait until the end of the episode to be able to calculate the total return.

However, in TD learning, we can leverage some of the learned properties to update the estimated values before reaching the end of the episode. This is called *bootstrapping* (in the context of RL, the term bootstrapping is not to be confused with the bootstrap estimates we used in *Chapter 7*, *Combining Different Models for Ensemble Learning*).

Similar to the dynamic programming approach and MC-based learning, we will consider two tasks: estimating the value function (which is also called value prediction) and improving the policy (which is also called the control task).

TD prediction

Let's first revisit the value prediction by MC. At the end of each episode, we are able to estimate the return G_t for each time step t. Therefore, we can update our estimates for the visited states as follows:

$$V(S_t) = V(S_t) + \alpha\big(G_t - V(S_t)\big)$$

Here, G_t is used as the *target return* to update the estimated values, and $(G_t - V(S_t))$ is a *correction* term added to our current estimate of the value $V(S_t)$. The value α is a hyperparameter denoting the learning rate, which is kept constant during learning.

Notice that in MC, the correction term uses the *actual* return, G_t, which is not known until the end of the episode. To clarify this, we can rename the actual return, G_t, to $G_{t:T}$, where the subscript $t{:}T$ indicates that this is the return obtained at time step t while considering all the events occurred from time step t until the final time step, T.

In TD learning, we replace the actual return, $G_{t:T}$, with a new target return, $G_{t:t+1}$, which significantly simplifies the updates for the value function, $V(S_t)$. The update-formula based on TD learning is as follows:

$$V(S_t) = V(S_t) + \alpha[G_{t:t+1} - V(S_t)]$$

Here, the target return, $G_{t:t+1} \overset{\text{def}}{=} R_{t+1} + \gamma V(S_{t+1}) = r + \gamma V(S_{t+1})$, is using the observed reward, $R_{t+1} = r$, and estimated value of the next immediate step. Notice the difference between MC and TD. In MC, $G_{t:T}$ is not available until the end of the episode, so we should execute as many steps as needed to get there. On the contrary, in TD, we only need to go one step ahead to get the target return. This is also known as TD(0).

Furthermore, the TD(0) algorithm can be generalized to the so-called *n-step TD* algorithm, which incorporates more future steps – more precisely, the weighted sum of *n* future steps. If we define *n* = 1, then the n-step TD procedure is identical to TD(0), which was described in the previous paragraph. If $n \rightarrow \infty$, however, the n-step TD algorithm will be the same as the MC algorithm. The update-rule for n-step TD is as follows:

$$V(S_t) = V(S_t) + \alpha[G_{t:t+n} - V(S_t)]$$

And $G_{t:t+n}$ is defined as:

$$G_{t:t+n} \stackrel{\text{def}}{=} \begin{cases} R_{t+1} + \gamma R_{t+2} + \cdots \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) & \text{if } t + n < T \\ G_{t:T} & \text{otherwise} \end{cases}$$

MC versus TD: which method converges faster?

While the precise answer to this question is still unknown, in practice, it is empirically shown that TD can converge faster than MC. If you are interested, you can find more details on the convergences of MC and TD in the book titled *Reinforcement Learning: An Introduction*, by Richard S. Sutton and Andrew G. Barto.

Now that we have covered the prediction task using the TD algorithm, we can move on to the control task. We will cover two algorithms for TD control: an *on-policy* control and an *off-policy* control. In both cases, we use the GPI that was used in both the dynamic programming and MC algorithms. In on-policy TD control, the value function is updated based on the actions from the same policy that the agent is following, while in an off-policy algorithm, the value function is updated based on actions outside the current policy.

On-policy TD control (SARSA)

For simplicity, we only consider the one-step TD algorithm, or TD(0). However, the on-policy TD control algorithm can be readily generalized to *n*-step TD. We will start by extending the prediction formula for defining the state-value function to describe the action-value function. To do this, we use a lookup table, that is, a tabular 2D-array, $Q(S_t, A_t)$, which represents the action-value function for each state-action pair. In this case, we will have the following:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

This algorithm is often called SARSA, referring to the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that is used in the update formula.

As we saw in the previous sections describing the dynamic programming and MC algorithms, we can use the GPI framework, and starting from the random policy, we can repeatedly estimate the action-value function for the current policy and then optimize the policy using the ϵ-greedy policy based on the current action-value function.

Off-policy TD control (Q-learning)

We saw when using the previous on-policy TD control algorithm that how we estimate the action-value function is based on the policy that is used in the simulated episode. After updating the action-value function, a separate step for policy improvement is performed by taking the action that has the higher value.

An alternative (and better) approach is to combine these two steps. In other words, imagine the agent is following policy π, generating an episode with the current transition quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Instead of updating the action-value function using the action value of A_{t+1} that is taken by the agent, we can find the best action even if it is not actually chosen by the agent following the current policy. (That's why this is considered an *off-policy* algorithm.)

To do this, we can modify the update rule to consider the maximum Q-value by varying different actions in the next immediate state. The modified equation for updating the Q-values is as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

We encourage you to compare the update rule here with that of the SARSA algorithm. As you can see, we find the best action in the next state, S_{t+1}, and use that in the correction term to update our estimate of $Q(S_t, A_t)$.

To put these materials into perspective, in the next section, we will see how to implement the Q-learning algorithm for solving the *grid world problem*.

Implementing our first RL algorithm

In this section, we will cover the implementation of the Q-learning algorithm to solve the *grid world problem*. To do this, we use the OpenAI Gym toolkit.

Introducing the OpenAI Gym toolkit

OpenAI Gym is a specialized toolkit for facilitating the development of RL models. OpenAI Gym comes with several predefined environments. Some basic examples are CartPole and MountainCar, where the tasks are to balance a pole and to move a car up a hill, respectively, as the names suggest. There are also many advanced robotics environments for training a robot to fetch, push, and reach for items on a bench or training a robotic hand to orient blocks, balls, or pens. Moreover, OpenAI Gym provides a convenient, unified framework for developing new environments. More information can be found on its official website: `https://gym.openai.com/`.

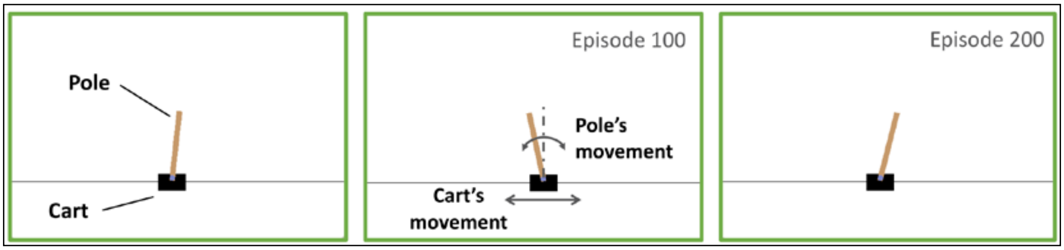
To follow the OpenAI Gym code examples in the next sections, you need to install the `gym` library, which can be easily done using `pip`:

```
> pip install gym
```

If you need additional help with the installation, please see the official installation guide at `https://gym.openai.com/docs/#installation`.

Working with the existing environments in OpenAI Gym

For practice with the Gym environments, let's create an environment from `CartPole-v1`, which already exists in OpenAI Gym. In this example environment, there is a pole attached to a cart that can move horizontally, as shown in the next figure:



The movements of the pole are governed by the laws of physics, and the goal for RL agents is to learn how to move the cart to stabilize the pole and prevent it from tipping over to either side.

Now, let's look at some properties of the CartPole environment in the context of RL, such as its state (or observation) space, action space, and how to execute an action:

```
>>> import gym
>>> env = gym.make('CartPole-v1')
>>> env.observation_space
Box(4,)
>>> env.action_space
Discrete(2)
```

In the preceding code, we created an environment for the `CartPole` problem. The observation space for this environment is `Box(4,)`, which represents a four-dimensional space corresponding to four real-valued numbers: the position of the cart, the cart's velocity, the angle of the pole, and the velocity of the tip of the pole. The action space is a discrete space, `Discrete(2)`, with two choices: pushing the cart either to the left or to the right.

The environment object, `env`, that we previously created by calling `gym.make('CartPole-v1')` has a `reset()` method that we can use to reinitialize an environment prior to each episode. Calling the `reset()` method will basically set the pole's starting state (S_0):

```
>>> env.reset()
array([-0.03908273, -0.00837535,  0.03277162, -0.0207195 ])
```

The values in the array returned by the `env.reset()` method call mean that the initial position of the cart is –0.039 with velocity –0.008, and the angle of the pole is 0.033 radian while the angular velocity of its tip is –0.021. Upon calling the `reset()` method, these values are initialized with random values with uniform distribution in the range [–0.05, 0.05].

After resetting the environment, we can interact with the environment by choosing an action and executing it by passing the action to the `step()` method:

```
>>> env.step(action=0)
(array([-0.03925023, -0.20395158,  0.03235723,  0.28212046]), 1.0,
False, {})
>>> env.step(action=1)
(array([-0.04332927, -0.00930575,  0.03799964, -0.00018409]), 1.0,
False, {})
```

Via the previous two commands, `env.step(action=0)` and `env.step(action=1)`, we pushed the cart to the left (`action=0`) and then to the right (`action=1`), respectively. Based on the selected action, the cart and its pole can move as governed by the laws of physics. Every time we call `env.step()`, it returns a tuple consisting of four elements:

- An array for the new state (or observations)
- A reward (a scalar value of type `float`)

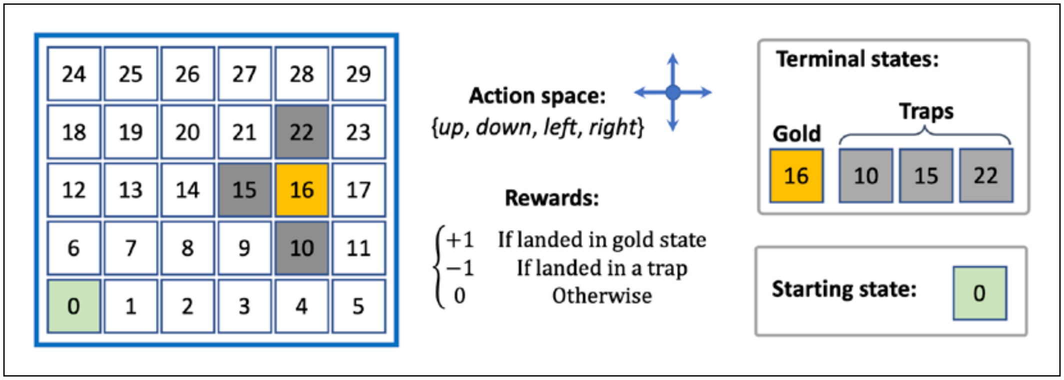
- A termination flag (`True` or `False`)
- A Python dictionary containing auxiliary information

The `env` object also has a `render()` method, which we can execute after each step (or a series of steps) to visualize the environment and the movements of the pole and cart, through time.

The episode terminates when the angle of the pole becomes larger than 12 degrees (from either side) with respect to an imaginary vertical axis, or when the position of the cart is more than 2.4 units from the center position. The reward defined in this example is to maximize the time the cart and pole are stabilized within the valid regions—in other words, the total reward (that is, return) can be maximized by maximizing the length of the episode.

A grid world example

After introducing the CartPole environment as a warm-up exercise for working with the OpenAI Gym toolkit, we will now switch to a different environment. We will work with a grid world example, which is a simplistic environment with *m* rows and *n* columns. Considering *m* = 4 and *n* = 6, we can summarize this environment as shown in the following figure:



In this environment, there are 30 different possible states. Four of these states are terminal states: a pot of gold at state 16 and three traps at states 10, 15, and 22. Landing in any of these four terminal states will end the episode, but with a difference between the gold and trap states. Landing on the gold state yields a positive reward, +1, whereas moving the agent onto one of the trap states yields a negative reward, –1. All other states have a reward of 0. The agent always starts from state 0. Therefore, every time we reset the environment, the agent will go back to state 0. The action space consists of four directions: move up, down, left, and right.

When the agent is at the outer boundary of the grid, selecting an action that would result in leaving the grid will not change the state.

Next, we will see how to implement this environment in Python, using the OpenAI Gym package.

Implementing the grid world environment in OpenAI Gym

For experimenting with the grid world environment via OpenAI Gym, using a script editor or IDE rather than executing the code interactively is highly recommended.

First, we create a new Python script named `gridworld_env.py` and then proceed by importing the necessary packages and two helper functions that we define for building the visualization of the environment.

In order to render the environments for visualization purposes, OpenAI Gym library uses the Pyglet library and provides wrapper classes and functions for our convenience. We will use these wrapper classes for visualizing the grid world environment in the following code example. More details about these wrapper classes can be found at

`https://github.com/openai/gym/blob/master/gym/envs/classic_control/rendering.py`

The following code example uses those wrapper classes:

```
## Script: gridworld_env.py

import numpy as np
from gym.envs.toy_text import discrete
from collections import defaultdict
import time
import pickle
import os

from gym.envs.classic_control import rendering

CELL_SIZE = 100
MARGIN = 10

def get_coords(row, col, loc='center'):
    xc = (col+1.5) * CELL_SIZE
    yc = (row+1.5) * CELL_SIZE
    if loc == 'center':
        return xc, yc
    elif loc == 'interior_corners':
        half_size = CELL_SIZE//2 - MARGIN
        xl, xr = xc - half_size, xc + half_size
        yt, yb = xc - half_size, xc + half_size
        return [(xl, yt), (xr, yt), (xr, yb), (xl, yb)]
    elif loc == 'interior_triangle':
        x1, y1 = xc, yc + CELL_SIZE//3
        x2, y2 = xc + CELL_SIZE//3, yc - CELL_SIZE//3
        x3, y3 = xc - CELL_SIZE//3, yc - CELL_SIZE//3
        return [(x1, y1), (x2, y2), (x3, y3)]

def draw_object(coords_list):
    if len(coords_list) == 1: # -> circle
        obj = rendering.make_circle(int(0.45*CELL_SIZE))
        obj_transform = rendering.Transform()
        obj.add_attr(obj_transform)
        obj_transform.set_translation(*coords_list[0])
        obj.set_color(0.2, 0.2, 0.2) # -> black
    elif len(coords_list) == 3: # -> triangle
        obj = rendering.FilledPolygon(coords_list)
        obj.set_color(0.9, 0.6, 0.2) # -> yellow
    elif len(coords_list) > 3: # -> polygon
        obj = rendering.FilledPolygon(coords_list)
        obj.set_color(0.4, 0.4, 0.8) # -> blue
    return obj
```

The first helper function, `get_coords()`, returns the coordinates of the geometric shapes that we will use to annotate the grid world environment, such as a triangle to display the gold or circles to display the traps. The list of coordinates is passed to `draw_object()`, which decides to draw a circle, a triangle, or a polygon based on the length of the input list of coordinates.

Now, we can define the grid world environment. In the same file (`gridworld_env.py`), we define a class named `GridWorldEnv`, which inherits from OpenAI Gym's `DiscreteEnv` class. The most important function of this class is the constructor method, `__init__()`, where we define the action space, specify the role of each action, and determine the terminal states (gold as well as traps) as follows:

```
    class GridWorldEnv(discrete.DiscreteEnv):
        def __init__(self, num_rows=4, num_cols=6, delay=0.05):
            self.num_rows = num_rows
            self.num_cols = num_cols

            self.delay = delay
```

```
move_up = lambda row, col: (max(row-1, 0), col)
move_down = lambda row, col: (min(row+1, num_rows-1), col)
move_left = lambda row, col: (row, max(col-1, 0))
move_right = lambda row, col: (
    row, min(col+1, num_cols-1))

self.action_defs={0: move_up, 1: move_right,
                  2: move_down, 3: move_left}

## Number of states/actions
nS = num_cols*num_rows
nA = len(self.action_defs)
self.grid2state_dict={(s//num_cols, s%num_cols):s
                      for s in range(nS)}
self.state2grid_dict={s:(s//num_cols, s%num_cols)
                      for s in range(nS)}

## Gold state
gold_cell = (num_rows//2, num_cols-2)

## Trap states
trap_cells = [((gold_cell[0]+1), gold_cell[1]),
               (gold_cell[0], gold_cell[1]-1),
               ((gold_cell[0]-1), gold_cell[1])]

gold_state = self.grid2state_dict[gold_cell]
trap_states = [self.grid2state_dict[(r, c)]
               for (r, c) in trap_cells]
self.terminal_states = [gold_state] + trap_states
print(self.terminal_states)

## Build the transition probability
P = defaultdict(dict)
for s in range(nS):
    row, col = self.state2grid_dict[s]
    P[s] = defaultdict(list)
    for a in range(nA):
        action = self.action_defs[a]
        next_s = self.grid2state_dict[action(row, col)]

        ## Terminal state
        if self.is_terminal(next_s):
            r = (1.0 if next_s == self.terminal_states[0]
                         else -1.0)
                else:
                    r = 0.0
                if self.is_terminal(s):
                    done = True
                    next_s = s
                else:
                    done = False
                P[s][a] = [(1.0, next_s, r, done)]

        ## Initial state distribution
        isd = np.zeros(nS)
        isd[0] = 1.0

        super(GridWorldEnv, self).__init__(nS, nA, P, isd)

        self.viewer = None
        self._build_display(gold_cell, trap_cells)

    def is_terminal(self, state):
        return state in self.terminal_states

    def _build_display(self, gold_cell, trap_cells):

        screen_width = (self.num_cols+2) * CELL_SIZE
        screen_height = (self.num_rows+2) * CELL_SIZE
        self.viewer = rendering.Viewer(screen_width,
                                       screen_height)

        all_objects = []

        ## List of border points' coordinates
        bp_list = [
            (CELL_SIZE-MARGIN, CELL_SIZE-MARGIN),
            (screen_width-CELL_SIZE+MARGIN, CELL_SIZE-MARGIN),
            (screen_width-CELL_SIZE+MARGIN,
             screen_height-CELL_SIZE+MARGIN),
            (CELL_SIZE-MARGIN, screen_height-CELL_SIZE+MARGIN)
        ]
        border = rendering.PolyLine(bp_list, True)
        border.set_linewidth(5)
        all_objects.append(border)

        ## Vertical lines
```

```
            for col in range(self.num_cols+1):
                x1, y1 = (col+1)*CELL_SIZE, CELL_SIZE
                x2, y2 = (col+1)*CELL_SIZE,\
                         (self.num_rows+1)*CELL_SIZE
                line = rendering.PolyLine([(x1, y1), (x2, y2)], False)
                all_objects.append(line)

            ## Horizontal lines
            for row in range(self.num_rows+1):
                x1, y1 = CELL_SIZE, (row+1)*CELL_SIZE
                x2, y2 = (self.num_cols+1)*CELL_SIZE,\
                         (row+1)*CELL_SIZE
                line=rendering.PolyLine([(x1, y1), (x2, y2)], False)
                all_objects.append(line)

            ## Traps: --> circles
            for cell in trap_cells:
                trap_coords = get_coords(*cell, loc='center')
                all_objects.append(draw_object([trap_coords]))

            ## Gold:  --> triangle
            gold_coords = get_coords(*gold_cell,
                                     loc='interior_triangle')
            all_objects.append(draw_object(gold_coords))

            ## Agent --> square or robot
            if (os.path.exists('robot-coordinates.pkl') and
                    CELL_SIZE==100):
                agent_coords = pickle.load(
                    open('robot-coordinates.pkl', 'rb'))
                starting_coords = get_coords(0, 0, loc='center')
                agent_coords += np.array(starting_coords)
            else:
                agent_coords = get_coords(
                    0, 0, loc='interior_corners')
            agent = draw_object(agent_coords)
            self.agent_trans = rendering.Transform()
            agent.add_attr(self.agent_trans)
            all_objects.append(agent)

            for obj in all_objects:
                self.viewer.add_geom(obj)

        def render(self, mode='human', done=False):
```

```
        if done:
            sleep_time = 1
        else:
            sleep_time = self.delay
        x_coord = self.s % self.num_cols
        y_coord = self.s // self.num_cols
        x_coord = (x_coord+0) * CELL_SIZE
        y_coord = (y_coord+0) * CELL_SIZE
        self.agent_trans.set_translation(x_coord, y_coord)
        rend = self.viewer.render(
             return_rgb_array=(mode=='rgb_array'))
        time.sleep(sleep_time)
        return rend

    def close(self):
        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

This code implements the grid world environment, from which we can create instances of this environment. We can then interact with it in a manner similar to that in the CartPole example. The implemented class, `GridWorldEnv`, inherits methods such as `reset()` for resetting the state and `step()` for executing an action. The details of the implementation are as follows:

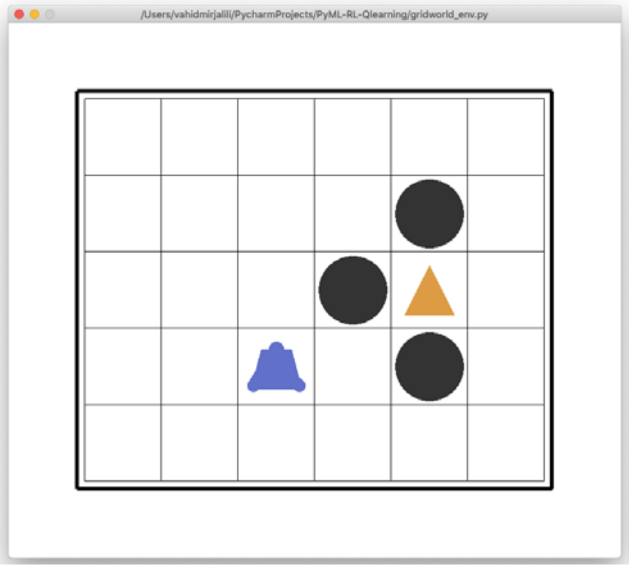
- We defined the four different actions using lambda functions: `move_up()`, `move_down()`, `move_left()`, and `move_right()`.
- The NumPy array `isd` holds the probabilities of the starting states so that a random state will be selected based on this distribution when the `reset()` method (from the parent class) is called. Since we always start from state 0 (the lower-left corner of the grid world), we set the probability of state 0 to 1.0 and the probabilities of all other 29 states to 0.0.
- The transition probabilities, defined in the Python dictionary `P`, determine the probabilities of moving from one state to another state when an action is selected. This allows us to have a probabilistic environment where taking an action could have different outcomes based on the stochasticity of the environment. For simplicity, we just use a single outcome, which is to change the state in the direction of the selected action. Finally, these transition probabilities will be used by the `env.step()` function to determine the next state.
- Furthermore, the function `_build_display()` will set up the initial visualization of the environment, and the `render()` function will show the movements of the agent.

Note that during the learning process, we do not know about the transition probabilities, and the goal is to learn through interacting with the environment. Therefore, we do not have access to `P` outside the class definition.

Now, we can test this implementation by creating a new environment and visualize a random episode by taking random actions at each state. Include the following code at the end of the same Python script (`gridworld_env.py`) and then execute the script:

```
if __name__ == '__main__':
    env = GridWorldEnv(5, 6)
    for i in range(1):
        s = env.reset()
        env.render(mode='human', done=False)

        while True:
            action = np.random.choice(env.nA)
            res = env.step(action)
            print('Action  ', env.s, action, ' -> ', res)
            env.render(mode='human', done=res[2])
            if res[2]:
                break

    env.close()
```

After executing the script, you should see a visualization of the grid world environment as depicted in the following figure:


/Users/vahidmirjalili/PycharmProjects/PyML-RL-Qlearning/gridworld_env.py

Solving the grid world problem with Q-learning

After focusing on the theory and the development process of RL algorithms, as well as setting up the environment via the OpenAI Gym toolkit, we will now implement the currently most popular RL algorithm, Q-learning. For this, we will use the grid world example that we already implemented in the script `gridworld_env.py`.

Implementing the Q-learning algorithm

Now, we create a new script and name it `agent.py`. In this `agent.py` script, we define an agent for interacting with the environment as follows:

```
## Script: agent.py

from collections import defaultdict
import numpy as np


class Agent(object):
    def __init__(
            self, env,
            learning_rate=0.01,
            discount_factor=0.9,
            epsilon_greedy=0.9,
            epsilon_min=0.1,
            epsilon_decay=0.95):
        self.env = env
        self.lr = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay

        ## Define the q_table
        self.q_table = defaultdict(lambda: np.zeros(self.env.nA))

    def choose_action(self, state):
        if np.random.uniform() < self.epsilon:
            action = np.random.choice(self.env.nA)
        else:
            q_vals = self.q_table[state]
            perm_actions = np.random.permutation(self.env.nA)
            q_vals = [q_vals[a] for a in perm_actions]
            perm_q_argmax = np.argmax(q_vals)
            action = perm_actions[perm_q_argmax]
        return action

    def _learn(self, transition):
        s, a, r, next_s, done = transition
        q_val = self.q_table[s][a]
        if done:
            q_target = r
        else:
            q_target = r + self.gamma*np.max(self.q_table[next_s])

        ## Update the q_table
        self.q_table[s][a] += self.lr * (q_target - q_val)

        ## Adjust the epislon
        self._adjust_epsilon()

    def _adjust_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

The `__init__()` constructor sets up various hyperparameters such as the learning rate, discount factor (Υ), and the parameters for the ϵ-greedy policy. Initially, we start with a high value of ϵ, but the method `_adjust_epsilon()` reduces it until it reaches the minimum value, ϵ_{min}. The method `choose_action()` chooses an action based on the ϵ-greedy policy as follows. A random uniform number is selected to determine whether the action should be selected randomly or otherwise, based on the action-value function. The method `_learn()` implements the update rule for the Q-learning algorithm. It receives a tuple for each transition, which consists of the current state (*s*), selected action (*a*), observed reward (*r*), next state (*s*'), as well as a flag to determine whether the end of the episode has been reached or not. The target value is equal to the observed reward (*r*) if this is flagged as end-of-episode; otherwise, the target is $r + \gamma \max_a Q(s', a)$.

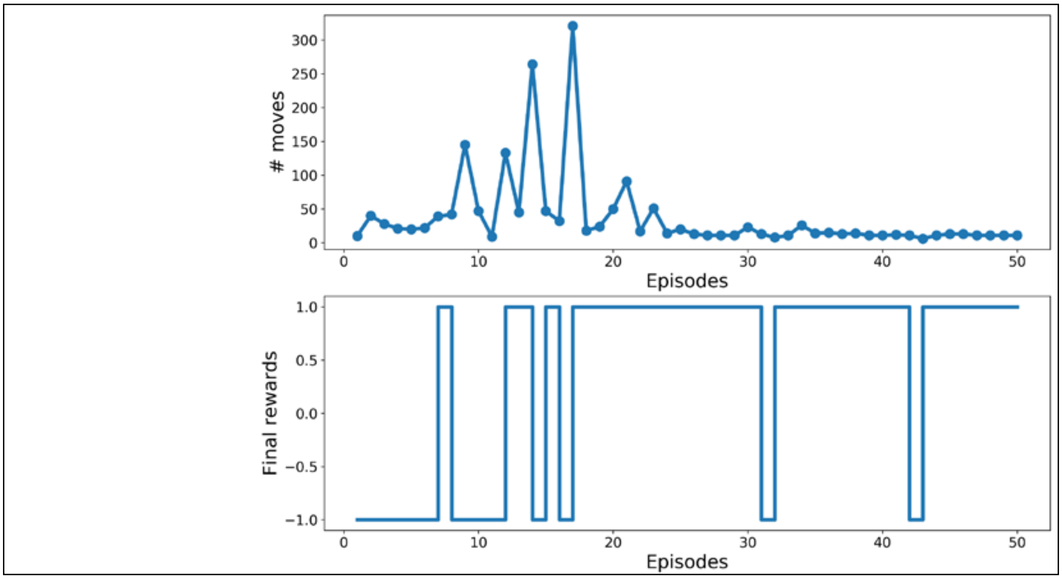
Finally, for our next step, we create a new script, `qlearning.py`, to put everything together and train the agent using the Q-learning algorithm.

In the following code, we define a function, `run_qlearning()`, that implements the Q-learning algorithm, simulating an episode by calling the `_choose_action()` method of the agent and executing the environment. Then, the transition tuple is passed to the `_learn()` method of the agent to update the action-value function. In addition, for monitoring the learning process, we also store the final reward of each episode (which could be –1 or +1), as well as the length of episodes (the number of moves taken by the agent from the start of the episode until the end).

The list of rewards and the number of moves is then plotted using the function `plot_learning_history()`:

```
## Script: qlearning.py

from gridworld_env import GridWorldEnv
from agent import Agent
from collections import namedtuple
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)

Transition = namedtuple(
    'Transition', ('state', 'action', 'reward',
                   'next_state', 'done'))

def run_qlearning(agent, env, num_episodes=50):
    history = []
    for episode in range(num_episodes):
        state = env.reset()
        env.render(mode='human')
        final_reward, n_moves = 0.0, 0
        while True:
            action = agent.choose_action(state)
            next_s, reward, done, _ = env.step(action)
            agent._learn(Transition(state, action, reward,
                                    next_s, done))
            env.render(mode='human', done=done)
            state = next_s
            n_moves += 1
            if done:
                break
            final_reward = reward
        history.append((n_moves, final_reward))
        print('Episode %d: Reward %.1f #Moves %d'
              % (episode, final_reward, n_moves))

    return history

def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 10))
    ax = fig.add_subplot(2, 1, 1)
    episodes = np.arange(len(history))
    moves = np.array([h[0] for h in history])
    plt.plot(episodes, moves, lw=4,
             marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('# moves', size=20)

    ax = fig.add_subplot(2, 1, 2)
    rewards = np.array([h[1] for h in history])
    plt.step(episodes, rewards, lw=4)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('Final rewards', size=20)
    plt.savefig('q-learning-history.png', dpi=300)
    plt.show()


if __name__ == '__main__':
    env = GridWorldEnv(num_rows=5, num_cols=6)
    agent = Agent(env)
    history = run_qlearning(agent, env)
    env.close()

    plot_learning_history(history)
```

Executing this script will run the Q-learning program for 50 episodes. The behavior of the agent will be visualized, and you can see that at the beginning of the learning process, the agent mostly ends up in the trap states. But through time, it learns from its failures and eventually finds the gold state (for instance, the first time in episode 7). The following figure shows the agent's number of moves and rewards:



The plotted learning history shown in the previous figure indicates that the agent, after 30 episodes, learns a short path to get to the gold state. As a result, the lengths of the episodes after the 30th episode are more or less the same, with minor deviations due to the ϵ-greedy policy.

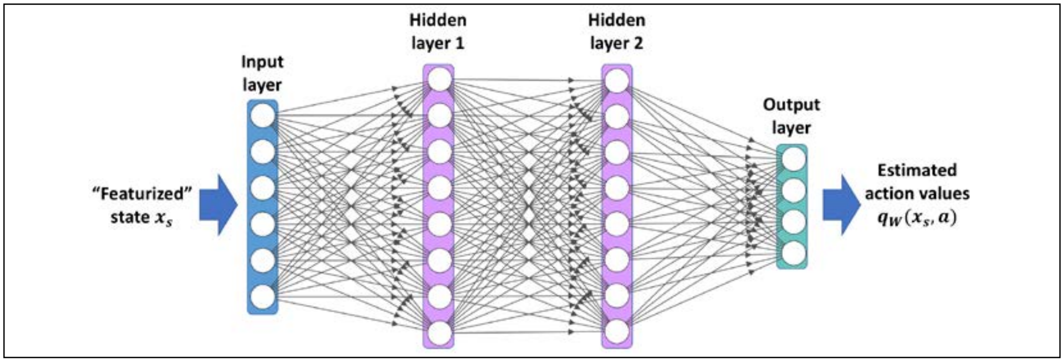
A glance at deep Q-learning

In the previous code, we saw an implementation of the popular Q-learning algorithm for the grid world example. This example consisted of a discrete state space of size 30, where it was sufficient to store the Q-values in a Python dictionary.

However, we should note that sometimes the number of states can get very large, possibly almost infinitely large. Also, we may be dealing with a continuous state space instead of working with discrete states. Moreover, some states may not be visited at all during training, which can be problematic when generalizing the agent to deal with such unseen states later.

To address these problems, instead of representing the value function in a tabular format like $V(S_t)$, or $Q(S_t, A_t)$, for the action-value function, we use a *function approximation* approach. Here, we define a parametric function, $v_W(x_s)$, that can learn to approximate the true value function, that is, $v_W(x_s) \approx v_\pi(s)$, where x_s is a set of input features (or "featurized" states).

When the approximator function, $q_W(x_s, a)$, is a deep neural network (DNN), the resulting model is called a **deep Q-network** (**DQN**). For training a DQN model, the weights are updated according to the Q-learning algorithm. An example of a DQN model is shown in the following figure, where the states are represented as features passed to the first layer:



Now, let's see how we can train a DQN using the *deep Q-learning* algorithm. Overall, the main approach is very similar to the tabular Q-learning method. The main difference is that we now have a multilayer NN that computes the action values.

Training a DQN model according to the Q-learning algorithm

In this section, we describe the procedure for training a DQN model using the Q-learning algorithm. The deep Q-learning approach requires us to make some modifications to our previously implemented standard Q-learning approach.

One such modification is in the agent's `choose_action()` method, which in the code of the previous section for Q-learning was simply accessing the action values stored in a dictionary. Now this function should be changed to perform a forward pass of the NN model for computing the action values.

The other modifications needed for the deep Q-learning algorithm are described in the following two subsections.

Replay memory

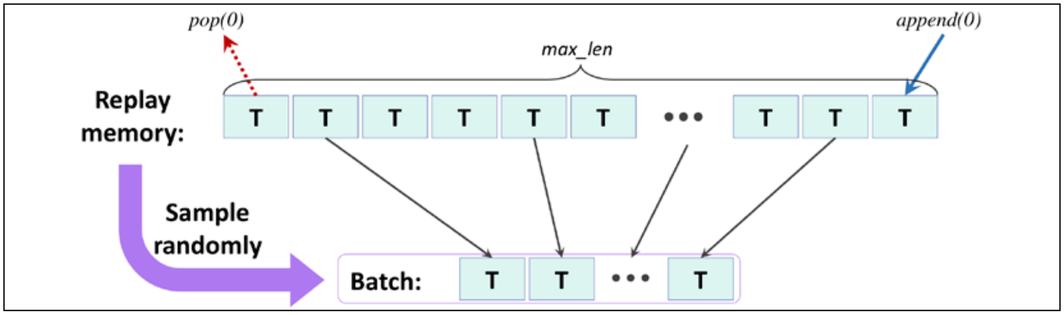
Using the previous tabular method for Q-learning, we could update the values for specific state-action pairs without affecting the values of others. However, now that we approximate $q(s, a)$ with an NN model, updating the weights for a state-action pair will likely affect the output of other states as well. When training NNs using stochastic gradient descent for a supervised task (for example, a classification task), we use multiple epochs to iterate through the training data multiple times until it converges.

This is not feasible in Q-learning, since the episodes will change during the training and as a result, some states that were visited in the early stages of training will become less likely to be visited later.

Furthermore, another problem is that when we train an NN, we assume that the training examples are **IID** (**independent and identically distributed**). However, the samples taken from an episode of the agent are not IID, as they obviously form a sequence of transitions.

To solve these issues, as the agent interacts with the environment and generates a transition quintuple $q_W(x_s, a)$, we store a large (but finite) number of such transitions in a memory buffer, often called *replay memory*. After each new interaction (that is, the agent selects an action and executes it in the environment), the resulting new transition quintuple is appended to the memory.

To keep the size of the memory bounded, the oldest transition will be removed from the memory (for example, if it is a Python list, we can use the `pop(0)` method to remove the first element of the list). Then, a mini-batch of examples is randomly selected from the memory buffer, which will be used for computing the loss and updating the network parameters. The following figure illustrates the process:



Implementing the replay memory

The replay memory can be implemented using a Python list, where every time we add a new element to the list, we need to check the size of the list and call `pop(0)` if needed.

Alternatively, we can use the `deque` data structure from the Python `collections` library, which allows us to specify an optional argument, `max_len`. By specifying the `max_len` argument, we will have a bounded deque. Therefore, when the object is full, appending a new element results in automatically removing an element from it.

Note that this is more efficient than using a Python list, since removing the first element of a list using `pop(0)` has O(n) complexity, while the deque's runtime complexity is O(1). You can learn more about the deque implementation from the official documentation that is available at

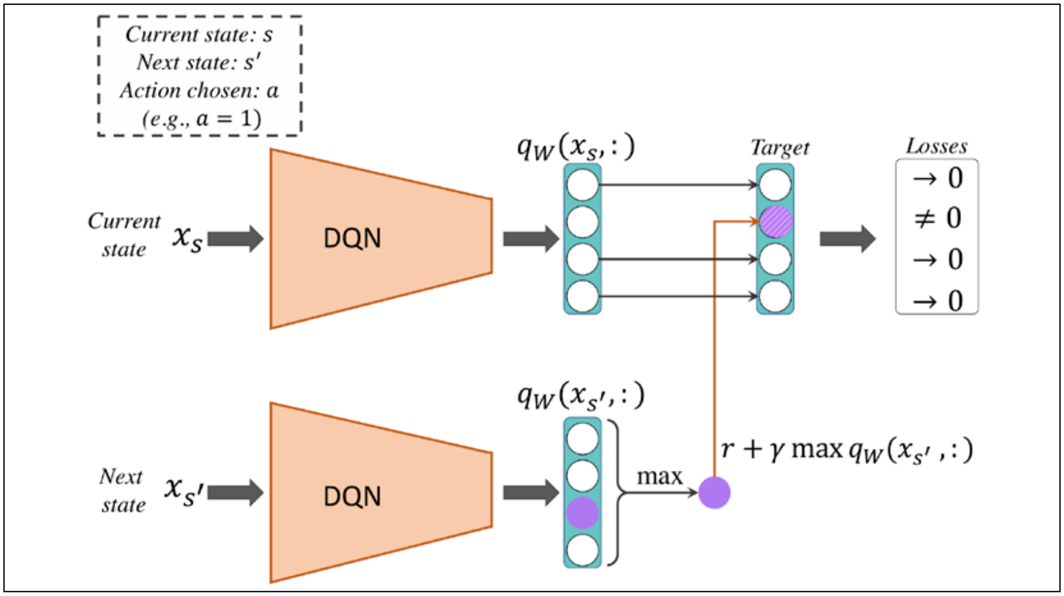
`https://docs.python.org/3.7/library/collections.html#collections.deque`.

Determining the target values for computing the loss

Another required change from the tabular Q-learning method is how to adapt the update rule for training the DQN model parameters. Recall that a transition quintuple, *T*, stored in the batch of examples, contains $(x_s, a, r, x_{s'}, done)$.

As shown in the following figure, we perform two forward passes of the DQN model. The first forward pass uses the features of the current state (x_s). Then, the second forward pass uses the features of the next state $(x_{s'})$. As a result, we will obtain the estimated action values, $q_W(x_s,:)$ and $q_W(x_{s'},:)$, from the first and second forward pass, respectively. (Here, this $q_W(x_s,:)$ notation means a vector of Q-values for all actions in $\hat{A}$.) From the transition quintuple, we know that action *a* is selected by the agent.

Therefore, according to the Q-learning algorithm, we need to update the action value corresponding to the state-action pair (x_s, a) with the scalar target value $r + \gamma \max_{a' \in \hat{A}} q_W(x_{s'}, a')$. Instead of forming a scalar target value, we will create a target action-value vector that retains the action values for other actions, $a' \neq a$, as shown in the following figure:



We treat this as a regression problem, using the following three quantities:

- The currently predicted values, $q_W(x_s, :)$
- The target value vector as described
- The standard mean squared error (MSE) cost function

As a result, the losses will be zero for every action except for *a*. Finally, the computed loss will be backpropagated to update the network parameters.

Implementing a deep Q-learning algorithm

Finally, we will use all these techniques for implementing a deep Q-learning algorithm. This time, we use the CartPole environment from the OpenAI gym environment that we introduced earlier. Recall that the CartPole environment has a continuous state space of size 4. In the following code, we define a class, `DQNAgent`, that builds the model and specifies various hyperparameters.

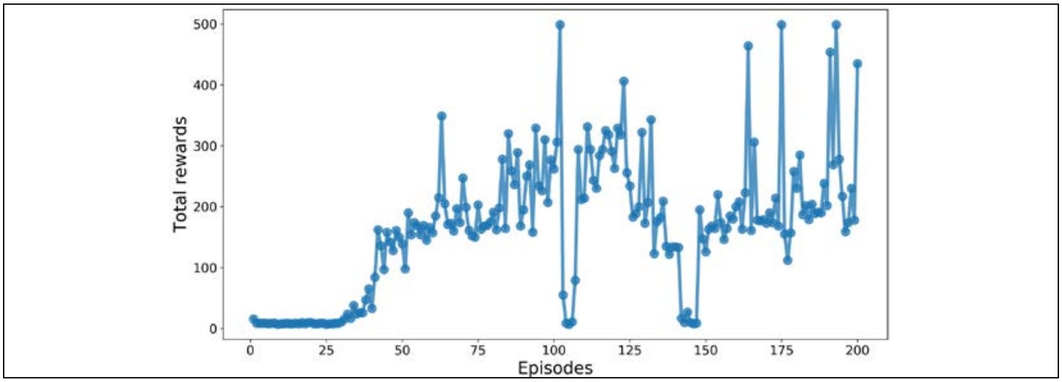
This class has two additional methods compared to the previous agent that was based on tabular Q-learning. The method `remember()` will append a new transition quintuple to the memory buffer, and the method `replay()` will create a mini-batch of example transitions and pass that to the `_learn()` method for updating the network's weight parameters:

```
import gym
import numpy as np
import tensorflow as tf
import random
import matplotlib.pyplot as plt
from collections import namedtuple
from collections import deque

np.random.seed(1)
tf.random.set_seed(1)

Transition = namedtuple(
            'Transition', ('state', 'action', 'reward',
                           'next_state', 'done'))

class DQNAgent:
    def __init__(
            self, env, discount_factor=0.95,
            epsilon_greedy=1.0, epsilon_min=0.01,
            epsilon_decay=0.995, learning_rate=1e-3,
            max_memory_size=2000):
        self.enf = env
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n

        self.memory = deque(maxlen=max_memory_size)

        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.lr = learning_rate
        self._build_nn_model()

    def _build_nn_model(self, n_layers=3):
        self.model = tf.keras.Sequential()

        ## Hidden layers
        for n in range(n_layers-1):
            self.model.add(tf.keras.layers.Dense(
                units=32, activation='relu'))
            self.model.add(tf.keras.layers.Dense(
                units=32, activation='relu'))

        ## Last layer
        self.model.add(tf.keras.layers.Dense(
            units=self.action_size))

        ## Build & compile model
        self.model.build(input_shape=(None, self.state_size))
        self.model.compile(
            loss='mse',
            optimizer=tf.keras.optimizers.Adam(lr=self.lr))

    def remember(self, transition):
        self.memory.append(transition)

    def choose_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        q_values = self.model.predict(state)[0]
        return np.argmax(q_values)  # returns action

    def _learn(self, batch_samples):
        batch_states, batch_targets = [], []
        for transition in batch_samples:
            s, a, r, next_s, done = transition
            if done:
                target = r
            else:
                target = (r +
                    self.gamma * np.amax(
                        self.model.predict(next_s)[0]
                    )
                )
            target_all = self.model.predict(s)[0]
            target_all[a] = target
            batch_states.append(s.flatten())
            batch_targets.append(target_all)
            self._adjust_epsilon()
        return self.model.fit(x=np.array(batch_states),
                              y=np.array(batch_targets),
                                 epochs=1,
                                 verbose=0)

    def _adjust_epsilon(self):
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def replay(self, batch_size):
        samples = random.sample(self.memory, batch_size)
        history = self._learn(samples)
        return history.history['loss'][0]
```

Finally, with the following code, we train the model for 200 episodes, and at the end visualize the learning history using the function `plot_learning_history()`:

```
def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 5))
    ax = fig.add_subplot(1, 1, 1)
    episodes = np.arange(len(history[0]))+1
    plt.plot(episodes, history[0], lw=4,
             marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('# Total Rewards', size=20)
    plt.show()

## General settings
EPISODES = 200
batch_size = 32
init_replay_memory_size = 500

if __name__ == '__main__':
    env = gym.make('CartPole-v1')
    agent = DQNAgent(env)
    state = env.reset()
    state = np.reshape(state, [1, agent.state_size])

    ## Filling up the replay-memory
    for i in range(init_replay_memory_size):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, agent.state_size])
        agent.remember(Transition(state, action, reward,
                                  next_state, done))
        if done:
            state = env.reset()
            state = np.reshape(state, [1, agent.state_size])
        else:
            state = next_state

    total_rewards, losses = [], []
    for e in range(EPISODES):
        state = env.reset()
        if e % 10 == 0:
            env.render()
        state = np.reshape(state, [1, agent.state_size])
        for i in range(500):
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.reshape(next_state,
                                    [1, agent.state_size])
            agent.remember(Transition(state, action, reward,
                                      next_state, done))
            state = next_state
            if e % 10 == 0:
                env.render()
            if done:
                total_rewards.append(i)
                print('Episode: %d/%d, Total reward: %d'
                      % (e, EPISODES, i))
                break
            loss = agent.replay(batch_size)
            losses.append(loss)
    plot_learning_history(total_rewards)
```

After training the agent for 200 episodes, we see that it indeed learned to increase the total rewards over time, as shown in the following plot:



Note that the total rewards obtained in an episode is equal to the amount of time that the agent is able to balance the pole. The learning history plotted in this figure shows that after about 30 episodes, the agent learns how to balance the pole and hold it for more than 200 time steps.

Chapter and book summary

In this chapter, we covered the essential concepts in RL, starting from the very foundations, and how RL can support decision making in complex environments.

We learned about agent-environment interactions and Markov decision processes (MDP), and we considered three main approaches for solving RL problems: dynamic programming, MC learning, and TD learning. We discussed that the dynamic programming algorithm assumes that the full knowledge of environment dynamics is available, an assumption that is not typically true for most real-world problems.

Then, we saw how the MC- and TD-based algorithms learn by allowing an agent to interact with the environment and generate a simulated experience. After discussing the underlying theory, we implemented the Q-learning algorithm as an off-policy subcategory of the TD algorithm for solving the grid world example. Finally, we covered the concept of function approximation and deep Q-learning in particular, which can be used for problems with large or continuous state spaces.

We hope you enjoyed this last chapter of *Python Machine Learning* and our exciting tour of machine learning and deep learning. Through the journey of this book, we've covered the essential topics that this field has to offer, and you should now be well equipped to put those techniques into action to solve real-world problems.

We started our journey with a brief overview of the different types of learning tasks: supervised learning, reinforcement learning, and unsupervised learning. We then discussed several different learning algorithms that you can use for classification, starting with simple single-layer NNs in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*.

We continued to discuss advanced classification algorithms in *Chapter 3*, *A Tour of Machine Learning Classifiers Using scikit-learn*, and we learned about the most important aspects of a machine learning pipeline in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Remember that even the most advanced algorithm is limited by the information in the training data that it gets to learn from. So, in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we learned about the best practices to build and evaluate predictive models, which is another important aspect in machine learning applications.

If one single learning algorithm does not achieve the performance we desire, it can be sometimes helpful to create an ensemble of experts to make a prediction. We explored this in *Chapter 7*, *Combining Different Models for Ensemble Learning*.

Then in *Chapter 8*, *Applying Machine Learning to Sentiment Analysis*, we applied machine learning to analyze one of the most popular and interesting forms of data in the modern age that's dominated by social media platforms on the Internet—text documents.

Next, we reminded ourselves that machine learning techniques are not limited to offline data analysis, and in *Chapter 9*, *Embedding a Machine Learning Model into a Web Application*, we saw how to embed a machine learning model into a web application to share it with the outside world.

For the most part, our focus was on algorithms for classification, which is probably the most popular application of machine learning. However, this is not where our journey ended! In *Chapter 10*, *Predicting Continuous Target Variables with Regression Analysis*, we explored several algorithms for regression analysis to predict continuous continuous target variables.

Another exciting subfield of machine learning is clustering analysis, which can help us find hidden structures in the data, even if our training data does not come with the right answers to learn from. We worked with this in *Chapter 11*, *Working with Unlabeled Data – Clustering Analysis*.

We then shifted our attention to one of one of the most exciting algorithms in the whole machine learning field—artificial neural networks. We started by implementing a multilayer perceptron from scratch with NumPy in *Chapter 12*, *Implementing a Multilayer Artificial Neural Network from Scratch*.

The utility of TensorFlow 2 for deep learning became obvious in *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, where we used TensorFlow to facilitate the process of building NN models and worked with TensorFlow `Dataset` objects, and learned how to apply preprocessing steps to a dataset.

We delved deeper into the mechanics of TensorFlow in *Chapter 14*, *Going Deeper – The Mechanics of TensorFlow*, and discussed the different aspects and mechanics of TensorFlow, including variables, TensorFlow function decoration, computing gradients of a computation, as well as TensorFlow estimators.

In *Chapter 15*, *Classifying Images with Deep Convolutional Neural Networks*, we dived into convolutional neural networks, which are widely used in computer vision at the moment, due to their great performance in image classification tasks.

In *Chapter 16*, *Modeling Sequential Data Using Recurrent Neural Networks*, we learned about sequence modeling using RNNs, and covered the Transformer model, one of the most recent deep learning algorithms for seq2seq modeling.

In *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*, we saw how to generate new images using GANs, and along the way, we also learned about autoencoders, batch normalization, transposed convolution, and Wasserstein GANs.

Finally, in this chapter, we covered a completely separate category of machine learning tasks and saw how to develop algorithms that learn by interacting with their environment through a reward process.

While a comprehensive study of deep learning is well beyond the scope of this book, we hope that we've kindled your interest enough to follow the most recent advancements in this field of deep learning.

If you're considering a career in machine learning, or you just want to keep up to date with the current advancements in this field, we can recommend to you the works of the following leading experts in the machine learning field:

- Geoffrey Hinton (`http://www.cs.toronto.edu/~hinton/`)
- Andrew Ng (`http://www.andrewng.org/`)
- Yann LeCun (`http://yann.lecun.com`)
- Juergen Schmidhuber (`http://people.idsia.ch/~juergen/`)
- Yoshua Bengio (`http://www.iro.umontreal.ca/~bengioy/yoshua_en/`)

Just to name a few!

Lastly, you can find out what we, the authors, are up to at these sites:

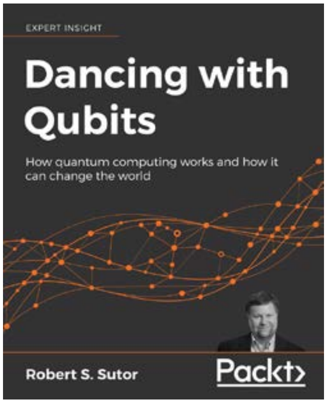
`https://sebastianraschka.com`

`http://vahidmirjalili.com`.

You're always welcome to contact us if you have any questions about this book or if you need some general tips about machine learning.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

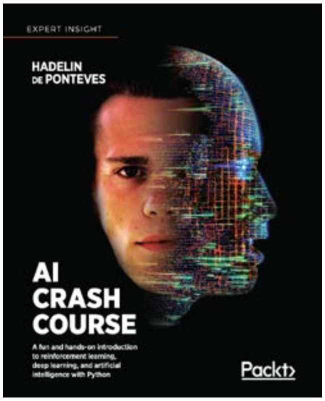


Dancing with Qubits

Robert S. Sutor

ISBN: 978-1-83882-736-6

- See how Quantum Computing works, what makes it different, and why it could be so powerful
- Discover the complex, mind-bending mechanics that underpin quantum systems
- Understand the necessary concepts behind classical and quantum computing
- Refresh and extend your grasp of computing, quantum theory, and quantum computing
- Explore the main applications of quantum computing to scientific computing, AI, and elsewhere
- Comprehend the detailed overview of qubits, quantum circuits, and quantum algorithm


EXPERT INSIGHT
HADELIN DE PONTEVES
AI CRASH COURSE
A fun and hands-on introduction to reinforcement learning, deep learning, and artificial intelligence with Python
Packt>

AI Crash Course

Hadelin de Ponteves

ISBN: 978-1-83864-535-9

- Roll up your sleeves and start programming AI models
- No math, data science, or machine learning background required
- Packed with hands-on examples, illustrations, and clear step-by-step instructions
- 5 hands-on working projects put ideas into action and show step-by-step how to build intelligent software

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index