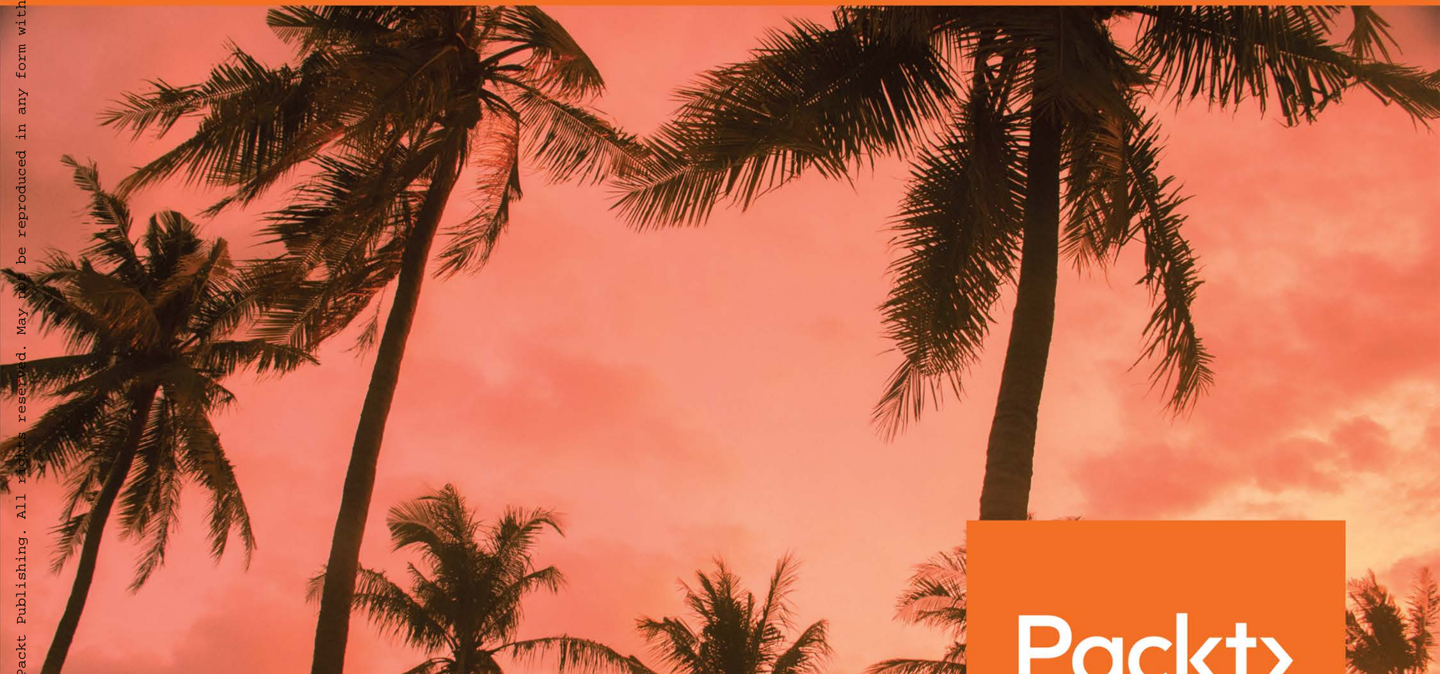
Big Data Analysis with Python

Combine Spark and Python to unlock the powers of parallel computing and machine learning



Ivan Marin, Ankit Shukla and Sarang VK

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/9/2023 10:53 AM via
AN: 2102158 ; Ivan Marin, Ankit Shukla, Sarang VK.; Big Data Analysis with Python : Combine Spark and Python to Unlock the Powers of Parallel Computing and Machine Learning
Account: ns335141

Copyright 2019. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

Big Data Analysis with Python

Combine Spark and Python to unlock the powers of parallel computing and machine learning

Ivan Marin, Ankit Shukla and Sarang VK

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Big Data Analysis with Python

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

Authors: Ivan Marin, Ankit Shukla and Sarang VK

Technical Reviewer: Namachivayam D

Managing Editor: Vishal Kamal Mewada

Acquisitions Editor: Kunal Sawant

Production Editor: Nitesh Thakur

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Shivangi Chatterji, Alex Mazonowicz, Douglas Paterson, Dominic Pereira,
Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur and Mohita Vyas

First Published: April 2019

Production Reference: 1080419

ISBN: 978-1-78995-528-6

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Table of Contents

Preface i

The Python Data Science Stack 1

Introduction 2
Python Libraries and Packages 2
- IPython: A Powerful Interactive Shell 3
- Exercise 1: Interacting with the Python Shell Using the IPython Commands 3
- The Jupyter Notebook 4
- Exercise 2: Getting Started with the Jupyter Notebook 6
- IPython or Jupyter? 9
- Activity 1: IPython and Jupyter 9
- NumPy 10
- SciPy 10
- Matplotlib 10
- Pandas 11

Using Pandas 12
- Reading Data 12
- Exercise 3: Reading Data with Pandas 12
- Data Manipulation 14
- **Selection and Filtering** 14
- **Selecting Rows Using Slicing** 15
- Exercise 4: Data Selection and the .loc Method 16
- **Applying a Function to a Column** 20
- Activity 2: Working with Data Problems 20

Data Type Conversion 21
- Exercise 5: Exploring Data Types 22

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Aggregation and Grouping 24
Exercise 6: Aggregation and Grouping Data 25
NumPy on Pandas 26
Exporting Data from Pandas 26
Exercise 7: Exporting Data in Different Formats 26
Visualization with Pandas 28
Activity 3: Plotting Data with Pandas 29
Summary 31

Statistical Visualizations 33

Introduction 34
Types of Graphs and When to Use Them 34
Exercise 8: Plotting an Analytical Function 35
Components of a Graph 37
Exercise 9: Creating a Graph 38
Exercise 10: Creating a Graph for a Mathematical Function 40
Seaborn 41
Which Tool Should Be Used? 41
Types of Graphs 42
Line Graphs 42
Time Series Plots 43
Exercise 11: Creating Line Graphs Using Different Libraries 44
Pandas DataFrames and Grouped Data 46
Activity 4: Line Graphs with the Object-Oriented API and Pandas DataFrames 47
Scatter Plots 48
Activity 5: Understanding Relationships of Variables Using Scatter Plots 50
Histograms 51
Exercise 12: Creating a Histogram of Horsepower Distribution 51

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Boxplots 54
Exercise 13: Analyzing the Behavior of the Number of Cylinders and Horsepower Using a Boxplot 55
Changing Plot Design: Modifying Graph Components 57
Title and Label Configuration for Axis Objects 58
Exercise 14: Configuring a Title and Labels for Axis Objects 58
Line Styles and Color 60
Figure Size 61
Exercise 15: Working with Matplotlib Style Sheets 62
Exporting Graphs 64
Activity 6: Exporting a Graph to a File on Disk 66
Activity 7: Complete Plot Design 67
Summary 68
Working with Big Data Frameworks 71
Introduction 72
Hadoop 73
Manipulating Data with the HDFS 74
Exercise 16: Manipulating Files in the HDFS 75
Spark 76
Spark SQL and Pandas DataFrames 77
Exercise 17: Performing DataFrame Operations in Spark 78
Exercise 18: Accessing Data with Spark 79
Exercise 19: Reading Data from the Local Filesystem and the HDFS 80
Exercise 20: Writing Data Back to the HDFS and PostgreSQL 82
Writing Parquet Files 82
Exercise 21: Writing Parquet Files 83
Increasing Analysis Performance with Parquet and Partitions 84
Exercise 22: Creating a Partitioned Dataset 85
Handling Unstructured Data 85

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 23: Parsing Text and Cleaning 86
Activity 8: Removing Stop Words from Text 88
Summary 89

Diving Deeper with Spark 91

Introduction 92
Getting Started with Spark DataFrames 92
Exercise 24: Specifying the Schema of a DataFrame 94
Exercise 25: Creating a DataFrame from an Existing RDD 95
Exercise 25: Creating a DataFrame Using a CSV File 95
Writing Output from Spark DataFrames 96
Exercise 27: Converting a Spark DataFrame to a Pandas DataFrame 97
Exploring Spark DataFrames 97
Exercise 28: Displaying Basic DataFrame Statistics 98
Activity 9: Getting Started with Spark DataFrames 100
Data Manipulation with Spark DataFrames 100
Exercise 29: Selecting and Renaming Columns from the DataFrame 101
Exercise 30: Adding and Removing a Column from the DataFrame 102
Exercise 31: Displaying and Counting Distinct Values in a DataFrame 103
Exercise 32: Removing Duplicate Rows and Filtering Rows of a DataFrame 103
Exercise 33: Ordering Rows in a DataFrame 105
Exercise 34: Aggregating Values in a DataFrame 106
Activity 10: Data Manipulation with Spark DataFrames 107
Graphs in Spark 107
Exercise 35: Creating a Bar Chart 108
Exercise 36: Creating a Linear Model Plot 109
Exercise 37: Creating a KDE Plot and a Boxplot 110
Activity 11: Graphs in Spark 112
Summary 114

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Handling Missing Values and Correlation Analysis 117

Introduction 118
Setting up the Jupyter Notebook 118
Missing Values 119
Exercise 38: Counting Missing Values in a DataFrame 120
Exercise 39: Counting Missing Values in All DataFrame Columns 120
Fetching Missing Value Records from the DataFrame 121
Handling Missing Values in Spark DataFrames 122
Exercise 40: Removing Records with Missing Values from a DataFrame 122
Exercise 41: Filling Missing Values with a Constant in a DataFrame Column 123
Correlation 124
Exercise 42: Computing Correlation 125
Activity 12: Missing Value Handling and Correlation Analysis with PySpark DataFrames 126
Summary 129

Exploratory Data Analysis 131

Introduction 132
Defining a Business Problem 132
Problem Identification 134
Requirement Gathering 134
Data Pipeline and Workflow 134
Identifying Measurable Metrics 135
Documentation and Presentation 135
Translating a Business Problem into Measurable Metrics and Exploratory Data Analysis (EDA) 136
Data Gathering 136
Analysis of Data Generation 136

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

KPI Visualization 137
Feature Importance 138
Exercise 43: Identify the Target Variable and Related KPIs from the Given Data for the Business Problem 138
Exercise 44: Generate the Feature Importance of the Target Variable and Carry Out EDA 146
Structured Approach to the Data Science Project Life Cycle 151
Data Science Project Life Cycle Phases 152
Phase 1: Understanding and Defining the Business Problem 152
Phase 2: Data Access and Discovery 152
Phase 3: Data Engineering and Pre-processing 153
Activity 13: Carry Out Mapping to Gaussian Distribution of Numeric Features from the Given Data 154
Phase 4: Model Development 155
Summary 155

Reproducibility in Big Data Analysis 157

Introduction 158
Reproducibility with Jupyter Notebooks 158
Introduction to the Business Problem 160
Documenting the Approach and Workflows 160
Explaining the Data Pipeline 160
Explain the Dependencies 161
Using Source Code Version Control 161
Modularizing the Process 161
Gathering Data in a Reproducible Way 162
Functionalities in Markdown and Code Cells 162
Explaining the Business Problem in the Markdown 164
Providing a Detailed Introduction to the Data Source 165
Explain the Data Attributes in the Markdown 165

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 45: Performing Data Reproducibility 167
Code Practices and Standards 170
Environment Documentation 171
Writing Readable Code with Comments 171
Effective Segmentation of Workflows 172
Workflow Documentation 173
Exercise 46: Missing Value Preprocessing with High Reproducibility 173
Avoiding Repetition 176
Using Functions and Loops for Optimizing Code 177
Developing Libraries/Packages for Code/Algorithm Reuse 178
Activity 14: Carry normalisation of data 178
Summary 179

Creating a Full Analysis Report 181

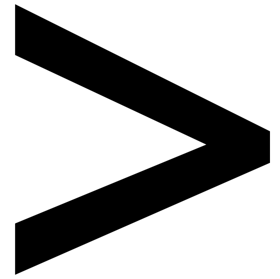
Introduction 182
Reading Data in Spark from Different Data Sources 182
Exercise 47: Reading Data from a CSV File Using the PySpark Object 182
Reading JSON Data Using the PySpark Object 184
SQL Operations on a Spark DataFrame 184
Exercise 48: Reading Data in PySpark and Carrying Out SQL Operations 185
Exercise 49: Creating and Merging Two DataFrames 189
Exercise 50: Subsetting the DataFrame 191
Generating Statistical Measurements 192
Activity 15: Generating Visualization Using Plotly 194
Summary 197

Appendix 199

Index 253

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use



Preface

About

This section briefly introduces the authors, the what this book covers, the technical skills you'll need to get started, and the hardware and software requirements required to complete the activities and exercises.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

About the Book

Processing big data in real time is challenging due to scalability, information inconsistency, and fault tolerance. Big Data Analysis with Python teaches you how to use tools that can control this data avalanche for you. With this book, you'll learn practical techniques to aggregate data into useful dimensions for posterior analysis, extract statistical measurements, and transform datasets into features for other systems.

The book begins with an introduction to data manipulation in Python using pandas. You'll then get familiar with statistical analysis and plotting techniques. With multiple hands-on activities in store, you'll be able to analyze data that is distributed on several computers by using Dask. As you progress, you'll study how to aggregate data for plots when the entire data cannot be accommodated in memory. You'll also explore Hadoop (HDFS and YARN), which will help you tackle larger datasets. The book also covers Spark and explains how it interacts with other tools.

By the end of this book, you'll be able to bootstrap your own Python environment, process large files, and manipulate data to generate statistics, metrics, and graphs.

About the Authors

Ivan Marin is a systems architect and data scientist working at Daitan Group, a Campinas-based software company. He designs big data systems for large volumes of data and implements machine learning pipelines end to end using Python and Spark. He is also an active organizer of data science, machine learning, and Python in São Paulo, and has given Python for data science courses at university level.

Ankit Shukla is a data scientist working with World Wide Technology, a leading US-based technology solution provider, where he develops and deploys machine learning and artificial intelligence solutions to solve business problems and create actual dollar value for clients. He is also part of the company's R&D initiative, which is responsible for producing intellectual property, building capabilities in new areas, and publishing cutting-edge research in corporate white papers. Besides tinkering with AI/ML models, he likes to read and is a big-time foodie.

Sarang VK is a lead data scientist at StraitsBridge Advisors, where his responsibilities include requirement gathering, solutioning, development, and productization of scalable machine learning, artificial intelligence, and analytical solutions using open source technologies. Alongside this, he supports pre-sales and competency.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Learning Objectives

- Use Python to read and transform data into different formats
- Generate basic statistics and metrics using data on disk
- Work with computing tasks distributed over a cluster
- Convert data from various sources into storage or querying formats
- Prepare data for statistical analysis, visualization, and machine learning
- Present data in the form of effective visuals

Approach

Big Data Analysis with Python takes a hands-on approach to understanding how to use Python and Spark to process data and make something useful out of it. It contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Audience

Big Data Analysis with Python is designed for Python developers, data analysts, and data scientists who want to get hands-on with methods to control data and transform it into impactful insights. Basic knowledge of statistical measurements and relational databases will help you to understand various concepts explained in this book.

Minimum Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

Processor: Intel or AMD 4-core or better

Memory: 8 GB RAM

Storage: 20 GB available space

Software Requirements

You'll need the following software installed in advance.

Any of the following operating systems:

- Windows 7 SP1 32/64-bit
- Windows 8.1 32/64-bit or Windows 10 32/64-bit
- Ubuntu 14.04 or later

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

- macOS Sierra or later
- Browser: Google Chrome or Mozilla Firefox
- Jupyter lab

You'll also need the following software installed in advance:

- Python 3.5+
- Anaconda 4.3+

These Python libraries are included with the Anaconda installation:

- matplotlib 2.1.0+
- iPython 6.1.0+
- requests 2.18.4+
- NumPy 1.13.1+
- pandas 0.20.3+
- scikit-learn 0.19.0+
- seaborn 0.8.0+
- bokeh 0.12.10+

These Python libraries require manual installation:

- mlxtend
- version_information
- ipython-sql
- pdir2
- graphviz

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To transform the data into the right data types, we can use conversion functions, such as **`to_datetime`**, **`to_numeric`**, and **`astype`**."

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

A block of code is set as follows:

```
before the sort function:[23, 66, 12, 54, 98, 3]
after the sort function: [3, 12, 23, 54, 66, 98]
```

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: "**Pandas** (https://pandas.pydata.org) is a data manipulation and analysis library that's widely used in the data science community."

Installation and Setup

Installing Anaconda:

1. Visit https://www.anaconda.com/download/ in your browser.
2. Click on Windows, Mac, or Linux, depending on the OS you are working on.
3. Next, click on the Download option. Make sure you download the latest version.
4. Open the installer after download.
5. Follow the steps in the installer and that's it! Your Anaconda distribution is ready.

PySpark is available on PyPi. To install PySpark run the following command:

```
pip install pyspark --upgrade
```

Updating Jupyter and installing dependencies:

1. Search for Anaconda Prompt and open it.
2. Type the following commands to update Conda and Jupyter:

   ```
   #Update conda
   conda update conda

   #Update Jupyter
   conda update Jupyter

   #install packages
   conda install numpy
   conda install pandas
   conda install statsmodels
   conda install matplotlib
   conda install seaborn
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. To open Jupyter Notebook from Anaconda Prompt, use the following command:

```
jupyter notebook
pip install -U scikit-learn
```

Installing the Code Bundle

Copy the code bundle for the class to the **C:/Code** folder.

Additional Resources

The code bundle for this book is also hosted on GitHub at: https://github.com/TrainingByPackt/Big-Data-Analysis-with-Python.

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

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

1

The Python Data Science Stack

Learning Objectives

We will start our journey by understanding the power of Python to manipulate and visualize data, creating useful analysis.

By the end of this chapter, you will be able to:

- Use all components of the Python data science stack
- Manipulate data using pandas DataFrames
- Create simple plots using pandas and Matplotlib

In this chapter, we will learn how to use NumPy, Pandas, Matplotlib, IPython, Jupyter notebook. Later in the chapter, we will explore how the deployment of **`virtualenv`**, **`pyenv`**, works, soon after that we will plot basic visualization using Matplotlib and Seaborn libraries.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

The Python data science stack is an informal name for a set of libraries used together to tackle data science problems. There is no consensus on which libraries are part of this list; it usually depends on the data scientist and the problem to be solved. We will present the libraries most commonly used together and explain how they can be used.

In this chapter, we will learn how to manipulate tabular data with the Python data science stack. The Python data science stack is the first stepping stone to manipulate large datasets, although these libraries are not commonly used for big data themselves. The ideas and the methods that are used here will be very helpful when we get to large datasets.

Python Libraries and Packages

One of the main reasons Python is a powerful programming language is the libraries and packages that come with it. There are more than 130,000 packages on the **Python Package Index** (**PyPI**) and counting! Let's explore some of the libraries and packages that are part of the data science stack.

The components of the data science stack are as follows:

- **NumPy**: A numerical manipulation package
- **pandas**: A data manipulation and analysis library
- **SciPy library**: A collection of mathematical algorithms built on top of NumPy
- **Matplotlib**: A plotting and graph library
- **IPython**: An interactive Python shell
- **Jupyter notebook**: A web document application for interactive computing

The combination of these libraries forms a powerful tool set for handling data manipulation and analysis. We will go through each of the libraries, explore their functionalities, and show how they work together. Let's start with the interpreters.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

IPython: A Powerful Interactive Shell

The IPython shell (https://ipython.org/) is an interactive Python command interpreter that can handle several languages. It allows us to test ideas quickly rather than going

through creating files and running them. Most Python installations have a bundled command interpreter, usually called the **shell**, where you can execute commands iteratively. Although it's handy, this standard Python shell is a bit cumbersome to use. IPython has more features:

- Input history that is available between sessions, so when you restart your shell, the previous commands that you typed can be reused.
- Using *Tab* completion for commands and variables, you can type the first letters of a Python command, function, or variable and IPython will autocomplete it.
- Magic commands that extend the functionality of the shell. Magic functions can enhance IPython functionality, such as adding a module that can reload imported modules after they are changed in the disk, without having to restart IPython.
- Syntax highlighting.

Exercise 1: Interacting with the Python Shell Using the IPython Commands

Getting started with the Python shell is simple. Let's follow these steps to interact with the IPython shell:

1. To start the Python shell, type the **`ipython`** command in the console:

   ```
   > ipython
   In [1]:
   ```

   The IPython shell is now ready and waiting for further commands. First, let's do a simple exercise to solve a sorting problem with one of the basic sorting methods, called **straight insertion**.

2. In the IPython shell, copy-paste the following code:

   ```
   import numpy as np

   vec = np.random.randint(0, 100, size=5)
   print(vec)
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now, the output for the randomly generated numbers will be similar to the following:

```
[23, 66, 12, 54, 98, 3]
```

3. Use the following logic to print the elements of the **vec** array in ascending order:

```
for j in np.arange(1, vec.size):
    v = vec[j]
    i = j
    while i > 0 and vec[i-1] > v:
        vec[i] = vec[i-1]
        i = i - 1
    vec[i] = v
```

Use the **print(vec)** command to print the output on the console:

```
[3, 12, 23, 54, 66, 98]
```

4. Now modify the code. Instead of creating an array of 5 elements, change its parameters so it creates an array with 20 elements, using the *up* arrow to edit the pasted code. After changing the relevant section, use the *down* arrow to move to the end of the code and press *Enter* to execute it.

Notice the number on the left, indicating the instruction number. This number always increases. We attributed the value to a variable and executed an operation on that variable, getting the result interactively. We will use IPython in the following sections.

The Jupyter Notebook

The Jupyter notebook (https://jupyter.org/) started as part of IPython but was separated in version 4 and extended, and lives now as a separate project. The notebook concept is based on the extension of the interactive shell model, creating documents that can run code, show documentation, and present results such as graphs and images.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Jupyter is a web application, so it runs in your web browser directly, without having to install separate software, and enabling it to be used across the internet. Jupyter can use IPython as a kernel for running Python, but it has support for more than 40 kernels that are contributed by the developer community.

> **Note**
>
> A kernel, in Jupyter parlance, is a computation engine that runs the code that is typed into a code cell in a notebook. For example, the IPython kernel executes Python code in a notebook. There are kernels for other languages, such as R and Julia.

It has become a de facto platform for performing operations related to data science from beginners to power users, and from small to large enterprises, and even academia. Its popularity has increased tremendously in the last few years. A Jupyter notebook contains both the input and the output of the code you run on it. It allows text, images, mathematical formulas, and more, and is an excellent platform for developing code and communicating results. Because of its web format, notebooks can be shared over the internet. It also supports the Markdown markup language and renders Markdown text as rich text, with formatting and other features supported.

As we've seen before, each notebook has a kernel. This kernel is the interpreter that will execute the code in the cells. The basic unit of a notebook is called a **cell**. A cell is a container for either code or text. We have two main types of cells:

- **Code cell**
- **Markdown cell**

A code cell accepts code to be executed in the kernel, displaying the output just below it. A Markdown cell accepts Markdown and will parse the text in Markdown to formatted text when the cell is executed.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's run the following exercise to get hands-on experience in the Jupyter notebook.

The fundamental component of a notebook is a cell, which can accept code or text depending on the mode that is selected.

Let's start a notebook to demonstrate how to work with cells, which have two states:

- Edit mode
- Run mode

When in edit mode, the contents of the cell can be edited, while in run mode, the cell is ready to be executed, either by the kernel or by being parsed to formatted text.

You can add a new cell by using the **Insert** menu option or using a keyboard shortcut, *Ctrl* + *B*. Cells can be converted between Markdown mode and code mode again using the menu or the Y shortcut key for a code cell and M for a Markdown cell.

To execute a cell, click on the **Run** option or use the *Ctrl* + *Enter* shortcut.

Exercise 2: Getting Started with the Jupyter Notebook

Let's execute the following steps to demonstrate how to start to execute simple programs in a Jupyter notebook.

Working with a Jupyter notebook for the first time can be a little confusing, but let's try to explore its interface and functionality. The reference notebook for this exercise is provided on GitHub.

Now, start a Jupyter notebook server and work on it by following these steps:

1. To start the Jupyter notebook server, run the following command on the console:

    ```
    > jupyter notebook
    ```

2. After successfully running or installing Jupyter, open a browser window and navigate to http://localhost:8888 to access the notebook.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. You should see a notebook similar to the one shown in the following screenshot:

Figure 1.1: Jupyter notebook

4. After that, from the top-right corner, click on **New** and select **Python 3** from the list.

5. A new notebook should appear. The first input cell that appears is a **Code** cell. The default cell type is **Code**. You can change it via the **Cell Type** option located under the **Cell** menu:

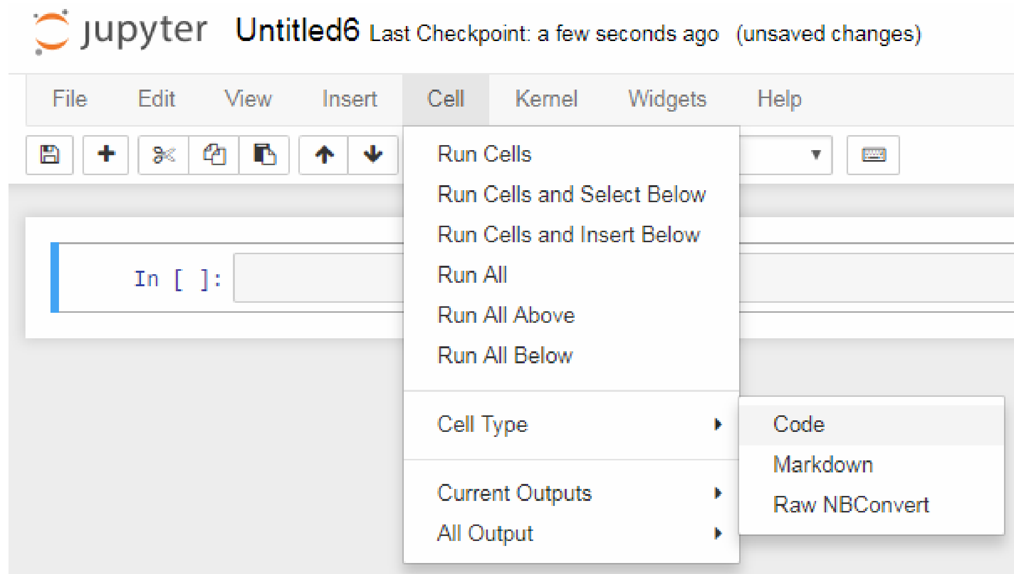


Figure 1.2: Options in the cell menu of Jupyter

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Now, in the newly generated **Code** cell, add the following arithmetic function in the first cell:

```
In []: x = 2
       print(x*2)
Out []: 4
```

7. Now, add a function that returns the arithmetic mean of two numbers, and then execute the cell:

```
In []: def mean(a,b):
       return (a+b)/2
```

8. Let's now use the **mean** function and call the function with two values, 10 and 20. Execute this cell. What happens? The function is called, and the answer is printed:

```
In []: mean(10,20)
Out[]: 15.0
```

9. We need to document this function. Now, create a new Markdown cell and edit the text in the Markdown cell, documenting what the function does:

The mean function calculates the arithmetic mean of two values.

This is a text cell. It accepts Markdown, Latex and HTML.

Figure 1.3: Markdown in Jupyter

10. Then, include an image from the web. The idea is that the notebook is a document that should register all parts of analysis, so sometimes we need to include a diagram or graph from other sources to explain a point.

11. Now, finally, include the mathematical expression in **LaTex** in the same Markdown cell:

$$f(x) = \int_{-\infty}^{\infty} e^{-2ikx} f(k)$$

Figure 1.4: LaTex expression in Jupyter Markdown

As we will see in the rest of the book, the notebook is the cornerstone of our analysis process. The steps that we just followed illustrate the use of different kinds of cells and the different ways we can document our analysis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

IPython or Jupyter?

Both IPython and Jupyter have a place in the analysis workflow. Usually, the IPython shell is used for quick interaction and more data-heavy work, such as debugging scripts or running asynchronous tasks. Jupyter notebooks, on the other hand, are great for presenting results and generating visual narratives with code, text, and figures. Most of the examples that we will show can be executed in both, except the graphical parts.

IPython is capable of showing graphs, but usually, the inclusion of graphs is more natural in a notebook. We will usually use Jupyter notebooks in this book, but the instructions should also be applicable to IPython notebooks.

Activity 1: IPython and Jupyter

Let's demonstrate common Python development in IPython and Jupyter. We will import NumPy, define a function, and iterate the results:

1. Open the **`python_script_student.py`** file in a text editor, copy the contents to a notebook in IPython, and execute the operations.
2. Copy and paste the code from the Python script into a Jupyter notebook.
3. Now, update the values of the **`x`** and **`c`** constants. Then, change the definition of the function.

> **Note**
>
> The solution for this activity can be found on page 200.

We now know how to handle functions and change function definitions on the fly in the notebook. This is very helpful when we are exploring and discovering the right approach for some code or an analysis. The iterative approach allowed by the notebook can be very productive in prototyping and faster than writing code to a script and executing that script, checking the results, and changing the script again.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

NumPy

NumPy (http://www.numpy.org) is a package that came from the Python scientific computing community. NumPy is great for manipulating multidimensional arrays and applying linear algebra functions to those arrays. It also has tools to integrate C, C++, and Fortran code, increasing its performance capabilities even more. There are a large number of Python packages that use NumPy as their numerical engine, including pandas and scikit-learn. These packages are part of SciPy, an ecosystem for packages used in mathematics, science, and engineering.

To import the package, open the Jupyter notebook used in the previous activity and type the following command:

```
import numpy as np
```

The basic NumPy object is **`ndarray`**, a homogeneous multidimensional array, usually composed of numbers, but it can hold generic data. NumPy also includes several functions for array manipulation, linear algebra, matrix operations, statistics, and other areas. One of the ways that NumPy shines is in scientific computing, where matrix and linear algebra operations are common. Another strength of NumPy is its tools that integrate with C++ and FORTRAN code. NumPy is also heavily used by other Python libraries, such as pandas.

SciPy

SciPy (https://www.scipy.org) is an ecosystem of libraries for mathematics, science, and engineering. NumPy, SciPy, scikit-learn, and others are part of this ecosystem. It is also the name of a library that includes the core functionality for lots of scientific areas.

Matplotlib

Matplotlib (https://matplotlib.org) is a plotting library for Python for 2D graphs. It's capable of generating figures in a variety of hard-copy formats for interactive use. It can use native Python data types, NumPy arrays, and pandas DataFrames as data sources. Matplotlib supports several backend–the part that supports the output generation in interactive or file format. This allows Matplotlib to be multiplatform. This flexibility also allows Matplotlib to be extended with toolkits that generate other kinds of plots, such as geographical plots and 3D plots.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The interactive interface for Matplotlib was inspired by the MATLAB plotting interface. It can be accessed via the **matplotlib.pyplot** module. The file output can write files directly to disk. Matplotlib can be used in scripts, in IPython or Jupyter environments, in web servers, and in other platforms. Matplotlib is sometimes considered low level because several lines of code are needed to generate a plot with more details. One of the tools that we will look at in this book that plots graphs, which are common in analysis, is the **Seaborn** library, one of the extensions that we mentioned before.

To import the interactive interface, use the following command in the Jupyter notebook:

```
import matplotlib.pyplot as plt
```

To have access to the plotting capabilities. We will show how to use Matplotlib in more detail in the next chapter.

Pandas

Pandas (https://pandas.pydata.org) is a data manipulation and analysis library that's widely used in the data science community. Pandas is designed to work with tabular or labeled data, similar to SQL tables and Excel files.

We will explore the operations that are possible with pandas in more detail. For now, it's important to learn about the two basic pandas data structures: the **series**, a unidimensional data structure; and the data science workhorse, the bi-dimensional **DataFrame**, a two-dimensional data structure that supports indexes.

Data in DataFrames and series can be ordered or unordered, homogeneous, or heterogeneous. Other great pandas features are the ability to easily add or remove rows and columns, and operations that SQL users are more familiar with, such as GroupBy, joins, subsetting, and indexing columns. Pandas is also great at handling time series data, with easy and flexible datetime indexing and selection.

Let's import pandas into the Jupyter notebook from the previous activity with the following command:

```
import pandas as pd
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Using Pandas

We will demonstrate the main operations for data manipulation using pandas. This approach is used as a standard for other data manipulation tools, such as Spark, so it's helpful to learn how to manipulate data using pandas. It's common in a big data pipeline to convert part of the data or a data sample to a pandas DataFrame to apply a more complex transformation, to visualize the data, or to use more refined machine learning models with the `scikit-learn` library. Pandas is also fast for in-memory, single-machine operations. Although there is a memory overhead between the data size and the pandas DataFrame, it can be used to manipulate large volumes of data quickly.

We will learn how to apply the basic operations:

- Read data into a DataFrame
- Selection and filtering
- Apply a function to data
- GroupBy and aggregation
- Visualize data from DataFrames

Let's start by reading data into a pandas DataFrame.

Reading Data

Pandas accepts several data formats and ways to ingest data. Let's start with the more common way, reading a CSV file. Pandas has a function called `read_csv`, which can be used to read a CSV file, either locally or from a URL. Let's read some data from the Socrata Open Data initiative, a RadNet Laboratory Analysis from the U.S. Environmental Protection Agency (EPA), which lists the radioactive content collected by the EPA.

Exercise 3: Reading Data with Pandas

How can an analyst start data analysis without data? We need to learn how to get data from an internet source into our notebook so that we can start our analysis. Let's demonstrate how pandas can read CSV data from an internet source so we can analyze it:

1. Import pandas library.

    ```
    import pandas as pd
    ```

2. Read the **Automobile mileage dataset**, available at this URL: https://github.com/TrainingByPackt/Big-Data-Analysis-with-Python/blob/master/Lesson01/imports-85.data. Convert it to csv.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Use the column names to name the data, with the parameter **names** on the **read_csv** function.

   ```
   Sample code : df = pd.read_csv("/path/to/imports-85.csv", names = columns)
   ```

4. Use the function **read_csv** from pandas and show the first rows calling the method **head** on the DataFrame:

   ```
   import pandas as pd
   df = pd.read_csv("imports-85.csv")
   df.head()
   ```

   The output is as follows:

	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.60	...	130	mpfi	3.47	2.68	9.00	111	5000	21	27	13495
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	16500
1	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19	26	16500
2	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24	30	13950
3	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18	22	17450
4	2	?	audi	gas	std	two	sedan	fwd	front	99.8	...	136	mpfi	3.19	3.40	8.5	110	5500	19	25	15250

Figure 1.5: Entries of the Automobile mileage dataset

Pandas can read more formats:

- JSON
- Excel
- HTML
- HDF5
- Parquet (with PyArrow)
- SQL databases
- Google Big Query

Try to read other formats from pandas, such as Excel sheets.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Data Manipulation

By data manipulation we mean any selection, transformation, or aggregation that is applied over the data. Data manipulation can be done for several reasons:

- To select a subset of data for analysis
- To clean a dataset, removing invalid, erroneous, or missing values
- To group data into meaningful sets and apply aggregation functions

Pandas was designed to let the analyst do these transformations in an efficient way.

Selection and Filtering

Pandas DataFrames can be sliced similarly to Python lists. For example, to select a subset of the first 10 rows of the DataFrame, we can use the `[0:10]` notation. We can see in the following screenshot that the selection of the interval `[1:3]` that in the NumPy representation selects the rows **1** and **2**.

```
df.head(10)
```

dff [1:3]

	State	Location	Date Posted	Date Collected	Sample Type	Unit	Ba-140	Co-60	Cs-134
0	ID	Boise	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
1	ID	Boise	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
2	AK	Juneau	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.0057
3	AK	Nome	03/30/2011	03/22/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
4	AK	Nome	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
5	AK	Nome	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.016
6	AK	Nome	04/04/2011	03/24/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.14
7	AK	Nome	04/04/2011	03/24/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.1
8	HI	Oahu	03/30/2011	03/20/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.034
9	HI	Oahu	03/30/2011	03/20/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.028

Figure 1.6: Selection in a pandas DataFrame

In the following section, we'll explore the selection and filtering operation in depth.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Selecting Rows Using Slicing

When performing data analysis, we usually want to see how data behaves differently under certain conditions, such as comparing a few columns, selecting only a few columns to help read the data, or even plotting. We may want to check specific values, such as the behavior of the rest of the data when one column has a specific value.

After selecting with slicing, we can use other methods, such as the **head** method, to select only a few rows from the beginning of the DataFrame. But how can we select some columns in a DataFrame?

To select a column, just use the name of the column. We will use the notebook. Let's select the **cylinders** column in our DataFrame using the following command:

```
df['State']
```

The output is as follows:

```
0    ID
1    ID
2    AK
3    AK
4    AK
Name: State, dtype: object
```

Figure 1.7: DataFrame showing the state

Another form of selection that can be done is filtering by a specific value in a column. For example, let's say that we want to select all rows that have the **State** column with the **MN** value. How can we do that? Try to use the Python equality operator and the DataFrame selection operation:

```
df[df.State == "MN"]
```

	State	Location	Date Posted	Date Collected
367	MN	St. Paul	04/08/2011	03/28/2011
368	MN	St. Paul	04/22/2011	04/13/2011
380	MN	Welch	04/08/2011	03/29/2011
381	MN	Welch	06/01/2011	04/14/2011
555	MN	St. Paul	04/04/2011	03/22/2011
556	MN	St. Paul	04/10/2011	03/29/2011

Figure 1.8: DataFrame showing the MN states

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

More than one filter can be applied at the same time. The **OR**, **NOT**, and **AND** logic operations can be used when combining more than one filter. For example, to select all rows that have **State** equal to **AK** and a **Location** of **Nome**, use the **&** operator:

```
df[(df.State == "AK") & (df.Location == "Nome")]
```

```
8]: df[(df.State == "AK") & (df.Location == "Nome")]
```

8]:

	State	Location	Date Posted	Date Collected	Sample Type	Unit	Ba-140	Co-60	Cs-134	Cs-136
3	AK	Nome	03/30/2011	03/22/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN	NaN
4	AK	Nome	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN	NaN
5	AK	Nome	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.016	NaN
6	AK	Nome	04/04/2011	03/24/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.14	Non-detect
7	AK	Nome	04/04/2011	03/24/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.1	0.012
18	AK	Nome	03/30/2011	03/22/2011	Air Cartridge	pCi/m3	Non-detect	Non-detect	NaN	NaN
19	AK	Nome	03/30/2011	03/23/2011	Air Cartridge	pCi/m3	Non-detect	Non-detect	NaN	NaN

df[(df.State == ‘AK’ & (df.Location == ‘Nome’))]

Figure 1.9: DataFrame showing State AK and Location Nome

Another powerful method is **.loc**. This method has two arguments, the row selection and the column selection, enabling fine-grained selection. An important caveat at this point is that, depending on the applied operation, the return type can be either a DataFrame or a series. The **.loc** method returns a series, as selecting only a column. This is expected, because each DataFrame column is a series. This is also important when more than one column should be selected. To do that, use two brackets instead of one, and use as many columns as you want to select.

Exercise 4: Data Selection and the .loc Method

As we saw before, selecting data, separating variables, and viewing columns and rows of interest is fundamental to the analysis process. Let's say we want to analyze the radiation from **I-131** in the state of **Minnesota**:

1. Import the NumPy and pandas libraries using the following command in the Jupyter notebook:

```
import numpy as np
import pandas as pd
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Read the RadNet dataset from the EPA, available from the Socrata project at https://github.com/TrainingByPackt/Big-Data-Analysis-with-Python/blob/master/Lesson01/RadNet_Laboratory_Analysis.csv:

```
url = "https://opendata.socrata.com/api/views/cf4r-dfwe/rows.
csv?accessType=DOWNLOAD"
df = pd.read_csv(url)
```

3. Start by selecting a column using the **['<name of the column>']** notation. Use the **State** column:

```
df['State'].head()
```

The output is as follows:

```
0    ID
1    ID
2    AK
3    AK
4    AK
Name: State, dtype: object
```

Figure 1.10: Data in the State column

4. Now filter the selected values in a column using the **MN** column name:

```
df[df.State == "MN"]
```

The output is as follows:

	State	Location	Date Posted	Date Collected	Sample Type	Unit	Ba-140	Co-60	Cs-134	Cs-136	Cs-137	I-131	I-132	I-133
367	MN	St. Paul	04/08/2011	03/28/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect
368	MN	St. Paul	04/22/2011	04/13/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	0.16	Non-detect	Non-detect
380	MN	Welch	04/08/2011	03/29/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect
381	MN	Welch	06/01/2011	04/14/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect
555	MN	St. Paul	04/04/2011	03/22/2011	Precipitation	pCi/l	Non-detect	Non-detect	Non-detect	NaN	Non-detect	32.3	Non-detect	Non-detect
556	MN	St. Paul	04/10/2011	03/29/2011	Precipitation	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	16	Non-detect	Non-detect
557	MN	Welch	04/04/2011	03/17/2011	Precipitation	pCi/l	Non-detect	Non-detect	Non-detect	NaN	Non-detect	Non-detect	Non-detect	Non-detect
558	MN	Welch/510	04/13/2011	04/04/2011	Precipitation	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	9.1	Non-detect	Non-detect

Figure 1.11: DataFrame showing States with MN

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Select more than one column per condition. Add the **Sample Type** column for filtering:

```
df[(df.State == 'CA') & (df['Sample Type'] == 'Drinking Water')]
```

The output is as follows:

	State	Location	Date Posted	Date Collected	Sample Type	Unit	Ba-140	Co-60	Cs-134	Cs-136	Cs-137	I-131	I-132	I-133	Te-129
305	CA	Los Angeles	04/10/2011	04/04/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	0.39	Non-detect	Non-detect	Non-detec
306	CA	Los Angeles	06/01/2011	04/12/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	0.18	Non-detect	Non-detect	Non-detec
356	CA	Richmond	04/09/2011	03/29/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detec
357	CA	Richmond	06/01/2011	04/13/2011	Drinking Water	pCi/l	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detect	Non-detec

Figure 1.12: DataFrame with State CA and Sample type as Drinking water

6. Next, select the **MN** state and the isotope **I-131**:

```
df[(df.State == "MN") ]["I-131"]
```

The output is as follows:

```
367    Non-detect
368          0.16
380    Non-detect
381    Non-detect
555          32.3
556            16
557    Non-detect
558           9.1
Name: I-131, dtype: object
```

Figure 1.13: Data showing the DataFrame with State Minnesota and Isotope I-131

The radiation in the state of Minnesota with ID **555** is the highest.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. We can do the same more easily with the `.loc` method, filtering by state and selecting a column on the same `.loc` call:

```
df_rad.loc[df_rad.State == "MN", "I-131"]
df[['I-132']].head()
```

The output is as follows:

	I-132
0	Non-detect
1	Non-detect
2	Non-detect
3	Non-detect
4	Non-detect

Figure 1.14: DataFrame with I-132

In this exercise, we learned how to filter and select values, either on columns or rows, using the NumPy slice notation or the `.loc` method. This can help when analyzing data, as we can check and manipulate only a subset of the data instead having to handle the entire dataset at the same time.

> **Note**
>
> The result of the `.loc` filter is a **series** and not a DataFrame. This depends on the
>
> operation and selection done on the DataFrame and not is caused only by `.loc`. Because the DataFrame can be understood as a 2D combination of series, the selection of one column will return a series. To make a selection and still return a DataFrame, use double brackets:
>
> ```
> df[['I-132']].head()
> ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Applying a Function to a Column

Data is never clean. There are always cleaning tasks that have to be done before a dataset can be analyzed. One of the most common tasks in data cleaning is applying a function to a column, changing a value to a more adequate one. In our example dataset, when no concentration was measured, the **`non-detect`** value was inserted. As this column is a numerical one, analyzing it could become complicated. We can apply a transformation over a column, changing from **`non-detect`** to **`numpy.NaN`**, which makes manipulating numerical values more easy, filling with other values such as the mean, and so on.

To apply a function to more than one column, use the **`applymap`** method, with the same logic as the **`apply`** method. For example, another common operation is removing spaces from strings. Again, we can use the **`apply`** and **`applymap`** functions to fix the data. We can also apply a function to rows instead of to columns, using the axis parameter (**`0`** for rows, **`1`** for columns).

Activity 2: Working with Data Problems

Before starting an analysis, we need to check for data problems, and when we find them (which is very common!), we have to correct the issues by transforming the DataFrame. One way to do that, for instance, is by applying a function to a column, or to the entire DataFrame. It's common for some numbers in a DataFrame, when it's read, to not be converted correctly to floating-point numbers. Let's fix this issue by applying functions:

1. Import **`pandas`** and **`numpy`** library.
2. Read the RadNet dataset from the U.S. Environmental Protection Agency.
3. Create a list with numeric columns for radionuclides in the RadNet dataset.
4. Use the **`apply`** method on one column, with a lambda function that compares the **`Non-detect`** string.
5. Replace the text values by **`NaN`** in one column with **`np.nan`**.
6. Use the same lambda comparison and use the **`applymap`** method on several columns at the same time, using the list created in the first step.
7. Create a list of the remaining columns that are not numeric.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Remove any spaces from these columns.
9. Using the selection and filtering methods, verify that the names of the string columns don't have any more spaces.

> **Note**
>
> The solution for this activity can be found on page 200.

The spaces in column names can be extraneous and can make selection and filtering more complicated. Fixing the numeric types helps when statistics must be computed using the data. If there is a value that is not valid for a numeric column, such as a string in a numeric column, the statistical operations will not work. This scenario happens, for example, when there is an error in the data input process, where the operator types the information by hand and makes mistakes, or the storage file was converted from one format to another, leaving incorrect values in the columns.

Data Type Conversion

Another common operation in data cleanup is getting the data types right. This helps with detecting invalid values and applying the right operations. The main types stored in pandas are as follows:

- **float** (**float64**, **float32**)
- **integer** (**int64**, **int32**)
- **datetime** (**datetime64[ns, tz]**)
- **timedelta** (**timedelta[ns]**)
- **bool**
- **object**
- **category**

Types can be set on, read, or inferred by pandas. Usually, if pandas cannot detect what data type the column is, it assumes that is object that stores the data as strings.

To transform the data into the right data types, we can use conversion functions such as **to_datetime**, **to_numeric**, or **astype**. Category types, columns that can only assume a limited number of options, are encoded as the **category** type.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 5: Exploring Data Types

Transform the data types in our example DataFrame to the correct types with the pandas **astype** function. Let's use the sample dataset from https://opendata.socrata.com/:

1. Import the required libraries, as illustrated here:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Read the data from the dataset as follows:

```
url = "https://opendata.socrata.com/api/views/cf4r-dfwe/rows.
csv?accessType=DOWNLOAD"
df = pd.read_csv(url)
```

3. Check the current data types using the **dtypes** function on the DataFrame:

```
df.dtypes
```

4. Use the **to_datetime** method to convert the dates from string format to **datetime** format:

```
df['Date Posted'] = pd.to_datetime(df['Date Posted'])
df['Date Collected'] = pd.to_datetime(df['Date Collected'])
columns = df.columns
id_cols = ['State', 'Location', "Date Posted", 'Date Collected', 'Sample
Type', 'Unit']
columns = list(set(columns) - set(id_cols))
columns
```

The output is as follows:

```
['Co-60',
 'Cs-136',
 'I-131',
 'Te-129',
 'Ba-140',
 'Cs-137',
 'Cs-134',
 'I-133',
 'I-132',
 'Te-132',
 'Te-129m']
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Use Lambda function:

```
df['Cs-134'] = df['Cs-134'].apply(lambda x: np.nan if x == "Non-detect"
else x)
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'Non-detect' else x)
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'ND' else x)
```

6. Apply the **to_numeric** method to the list of numeric columns created in the previous activity to convert the columns to the correct numeric types:

```
for col in columns:
    df[col] = pd.to_numeric(df[col])
```

7. Check the types of the columns again. They should be **float64** for the numeric columns and **datetime64[ns]** for the date columns:

```
df.dypes
```

8. Use the **astype** method to transform the columns that are not numeric to the **category** type:

```
df['State'] = df['State'].astype('category')
df['Location'] = df['Location'].astype('category')
df['Unit'] = df['Unit'].astype('category')
df['Sample Type'] = df['Sample Type'].astype('category')
```

9. Check the types with the **dtype** function for the last time:

```
df.dtypes
```

The output is as follows:

```
State                  category
Location               category
Date Posted      datetime64[ns]
Date Collected   datetime64[ns]
Sample Type            category
Unit                   category
Ba-140                  float64
Co-60                   float64
Cs-134                  float64
Cs-136                  float64
Cs-137                  float64
I-131                   float64
I-132                   float64
I-133                   float64
Te-129                  float64
Te-129m                 float64
Te-132                  float64
dtype: object
```

Figure 1.15: DataFrame and its types

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now our dataset looks fine, with all values correctly cast to the right types. But correcting the data is only part of the story. We want, as analysts, to understand the data from different perspectives. For example, we may want to know which state has the most contamination, or the radionuclide that is the least prevalent across cities. We may ask about the number of valid measurements present in the dataset. All these questions have in common transformations that involve grouping data together and aggregating several values. With pandas, this is accomplished with `GroupBy`. Let's see how we can use it by key and aggregate the data.

Aggregation and Grouping

After getting the dataset, our analyst may have to answer a few questions. For example, we know the value of the radionuclide concentration per city, but an analyst may be asked to answer: which state, on average, has the highest radionuclide concentration?

To answer the questions posed, we need to group the data somehow and calculate an aggregation on it. But before we go into grouping data, we have to prepare the dataset so that we can manipulate it in an efficient manner. Getting the right types in a pandas DataFrame can be a huge boost for performance and can be leveraged to enforce data consistency– it makes sure that numeric data really is numeric and allows us to execute operations that we want to use to get the answers.

`GroupBy` allows us to get a more general view of a feature, arranging data given a `GroupBy` key and an aggregation operation. In pandas, this operation is done with the `GroupBy` method, over a selected column, such as State. Note the aggregation operation after the `GroupBy` method. Some examples of the operations that can be applied are as follows:

- `mean`
- `median`
- `std` (**standard deviation**)
- `mad` (**mean absolute deviation**)
- `sum`
- `count`
- `abs`

> **Note**
>
> Several statistics, such as **mean** and **standard deviation**, only make sense with numeric data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

After applying **GroupBy**, a specific column can be selected and the aggregation operation can be applied to it, or all the remaining columns can be aggregated by the same function. Like SQL, **GroupBy** can be applied to more than one column at a time, and more than one aggregation operation can be applied to selected columns, one operation per column.

The **GroupBy** command in Pandas has some options, such as **as_index**, which can override the standard of transforming grouping key's columns to indexes and leaving them as normal columns. This is helpful when a new index will be created after the **GroupBy** operation, for example.

Aggregation operations can be done over several columns and different statistical methods at the same time with the **agg** method, passing a dictionary with the name of the column as the **key** and a list of statistical operations as **values**.

Exercise 6: Aggregation and Grouping Data

Remember that we have to answer the question of which state has, on average, the highest radionuclide concentration. As there are several cities per state, we have to combine the values of all cities in one state and calculate the average. This is one of the applications of **GroupBy**: calculating the average values of one variable as per a grouping. We can answer the question using **GroupBy**:

1. Import the required libraries:

   ```
   import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt
   import seaborn as sns
   ```

2. Load the datasets from the https://opendata.socrata.com/:

   ```
   df = pd.read_csv('RadNet_Laboratory_Analysis.csv')
   ```

3. Group the DataFrame using the **State** column.

   ```
   df.groupby('State')
   ```

4. Select the radionuclide **Cs-134** and calculate the average value per group:

   ```
   df.groupby('State')['Cs-134'].head()
   ```

5. Do the same for all columns, grouping per state and applying directly the **mean** function:

   ```
   df.groupby('State').mean().head()
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Now, group by more than one column, using a list of grouping columns.
7. Aggregate using several aggregation operations per column with the **agg** method. Use the **State** and **Location** columns:

```
df.groupby(['State', 'Location']).agg({'Cs-134':['mean', 'std'],
'Te-129':['min', 'max']})
```

NumPy on Pandas

NumPy functions can be applied to DataFrames directly or through the **apply** and **applymap** methods. Other NumPy functions, such as **np.where**, also work with DataFrames.

Exporting Data from Pandas

After creating an intermediate or final dataset in pandas, we can export the values from the DataFrame to several other formats. The most common one is CSV, and the command to do so is **df.to_csv('filename.csv')**. Other formats, such as Parquet and JSON, are also supported.

> **Note**
>
> Parquet is particularly interesting, and it is one of the big data formats that we will discuss later in the book.

Exercise 7: Exporting Data in Different Formats

After finishing our analysis, we may want to save our transformed dataset with all the corrections, so if we want to share this dataset or redo our analysis, we don't have to transform the dataset again. We can also include our analysis as part of a larger data pipeline or even use the prepared data in the analysis as input to a machine learning algorithm. We can accomplish data exporting our DataFrame to a file with the right format:

1. Import all the required libraries and read the data from the dataset using the following command:

```
import numpy as np
import pandas as pd

url = "https://opendata.socrata.com/api/views/cf4r-dfwe/rows.
csv?accessType=DOWNLOAD"
df = pd.read_csv(url)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Redo all adjustments for the data types (date, numeric, and categorical) in the RadNet data. The type should be the same as in *Exercise 6: Aggregation and Grouping Data*.

2. Select the numeric columns and the categorical columns, creating a list for each of them:

```
columns = df.columns
id_cols = ['State', 'Location', "Date Posted", 'Date Collected', 'Sample
Type', 'Unit']
columns = list(set(columns) - set(id_cols))
columns
```

The output is as follows:

```
['Cs-134',
 'Cs-136',
 'I-132',
 'Te-132',
 'Ba-140',
 'Te-129m',
 'Te-129',
 'I-133',
 'Cs-137',
 'Co-60',
 'I-131']
```

Apply the lambda replacing "Non-detect" by np.nan:

Figure 1.16: List of columns

3. Apply the lambda function that replaces **`Non-detect`** with **`np.nan`**:

```
df['Cs-134'] = df['Cs-134'].apply(lambda x: np.nan if x == "Non-detect"
else x)
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'Non-detect' else x)
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'ND' else x)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Remove the spaces from the categorical columns:

```
df.loc[:, ['State', 'Location', 'Sample Type', 'Unit']] = df.loc[:,
['State', 'Location', 'Sample Type', 'Unit']].applymap(lambda x:
x.strip())
```

5. Transform the date columns to the **`datetime`** format:

```
df['Date Posted'] = pd.to_datetime(df['Date Posted'])
df['Date Collected'] = pd.to_datetime(df['Date Collected'])
```

6. Transform all numeric columns to the correct numeric format with the **`to_numeric`** method:

```
for col in columns:
    df[col] = pd.to_numeric(df[col])
```

7. Transform all categorical variables to the **`category`** type:

```
df['State'] = df['State'].astype('category')
df['Location'] = df['Location'].astype('category')
df['Unit'] = df['Unit'].astype('category')
df['Sample Type'] = df['Sample Type'].astype('category')
```

8. Export our transformed DataFrame, with the right values and columns, to the CSV format with the **`to_csv`** function. Exclude the index using **`index=False`**, use a semicolon as the separator **`sep=";"`**, and encode the data as UTF-8 **`encoding="utf-8"`**:

```
df.to_csv('radiation_clean.csv', index=False, sep=';', encoding='utf-8')
```

9. Export the same DataFrame to the Parquet columnar and binary format with the **`to_parquet`** method:

```
df.to_parquet('radiation_clean.prq', index=False)
```

> **Note**
>
> Be careful when converting a datetime to a string!

Visualization with Pandas

Pandas can be thought as a data Swiss Army knife, and one thing that a data scientist always needs when analyzing data is to visualize that data. We will go into detail on the kinds of plot that we can apply in an analysis. For now, the idea is to show how to do **quick and dirty** plots directly from pandas.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The **plot** function can be called directly from the DataFrame selection, allowing fast visualizations. A scatter plot can be created by using Matplotlib and passing data from the DataFrame to the plotting function. Now that we know the tools, let's focus on the pandas interface for data manipulation. This interface is so powerful that it is replicated by other projects that we will see in this course, such as Spark. We will explain the plot components and methods in more detail in the next chapter.

You will see how to create graphs that are useful for statistical analysis in the next chapter. Focus here on the mechanics of creating plots from pandas for quick visualizations.

Activity 3: Plotting Data with Pandas

To finish up our activity, let's redo all the previous steps and plot graphs with the results, as we would do in a preliminary analysis:

1. Use the RadNet DataFrame that we have been working with.
2. Fix all the data type problems, as we saw before.
3. Create a plot with a filter per **Location**, selecting the city of **San Bernardino**, and one radionuclide, with the x-axis as date and the y-axis as radionuclide **I-131**:

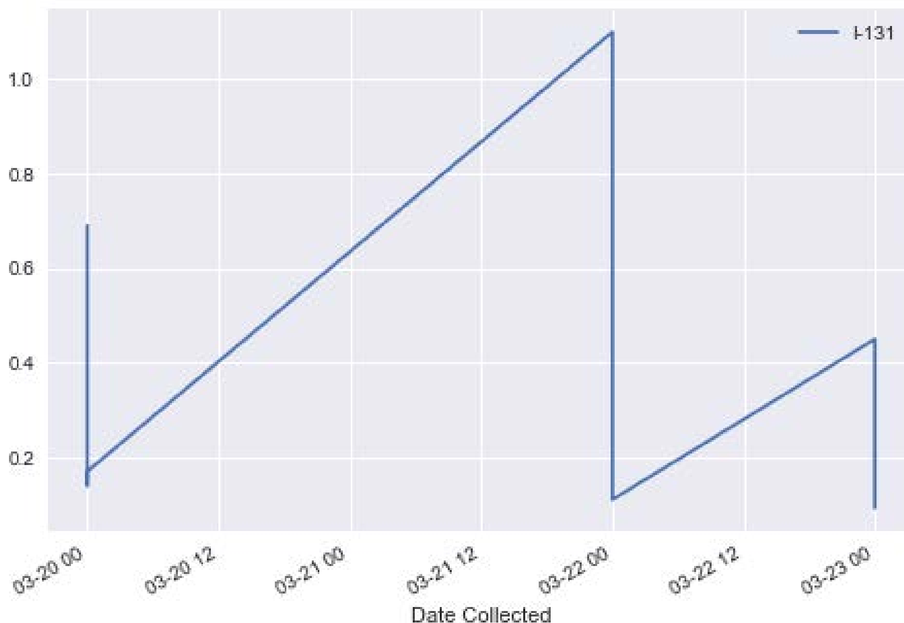


Figure 1.17: Plot of Location with I-131

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Create a scatter plot with the concentration of two related radionuclides, **`I-131`** and **`I-132`**:

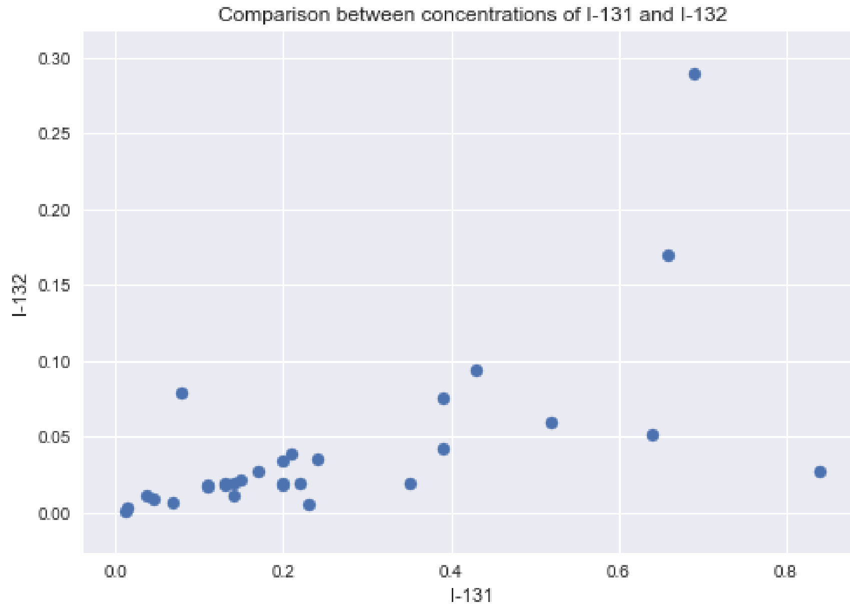


Figure 1.18: Plot of I-131 and I-132

> **Note**
>
> The solution for this activity can be found on page 203.

We are getting a bit ahead of ourselves here with the plotting, so we don't need to worry about the details of the plot or how we attribute titles, labels, and so on. The important takeaway here is understanding that we can plot directly from the DataFrame for quick analysis and visualization.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

We have learned about the most common Python libraries used in data analysis and data science, which make up the Python data science stack. We learned how to ingest data, select it, filter it, and aggregate it. We saw how to export the results of our analysis and generate some quick graphs.

These are steps done in almost any data analysis. The ideas and operations demonstrated here can be applied to data manipulation with big data. Spark DataFrames were created with the pandas interface in mind, and several operations are performed in a very similar fashion in pandas and Spark, greatly simplifying the analysis process. Another great advantage of knowing your way around pandas is that Spark can convert its DataFrames to pandas DataFrames and back again, enabling analysts to work with the best tool for the job.

Before going into big data, we need to understand how to better visualize the results of our analysis. Our understanding of the data and its behavior can be greatly enhanced if we visualize it using the correct plots. We can draw inferences and see anomalies and patterns when we plot the data.

In the next chapter, we will learn how to choose the right graph for each kind of data and analysis, and how to plot it using Matplotlib and Seaborn.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2

Statistical Visualizations

Learning Objectives

We will start our journey by understanding the power of Python to manipulate and visualize data, creating useful analysis.

By the end of this chapter, you will be able to:

- Use graphs for data analysis
- Create graphs of various types
- Change graph parameters such as color, title, and axis
- Export graphs for presentation, printing, and other uses

In this chapter, we will illustrate how the students can generate visualizations with Matplotlib and Seaborn.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

In the last chapter, we learned that the libraries that are most commonly used for data science work with Python. Although they are not big data libraries per se, the libraries of the Python Data Science Stack (**NumPy**, **Jupyter**, **IPython**, **Pandas**, and **Matplotlib**) are important in big data analysis.

As we will demonstrate in this chapter, no analysis is complete without visualizations, even with big datasets, so knowing how to generate images and graphs from data in Python is relevant for our goal of big data analysis. In the subsequent chapters, we will demonstrate how to process large volumes of data and aggregate it to visualize it using Python tools.

There are several visualization libraries for Python, such as Plotly, Bokeh, and others. But one of the oldest, most flexible, and most used is Matplotlib. But before going through the details of creating a graph with Matplotlib, let's first understand what kinds of graphs are relevant for analysis.

Types of Graphs and When to Use Them

Every analysis, whether on small or large datasets, involves a descriptive statistics step, where the data is summarized and described by statistics such as mean, median, percentages, and correlation. This step is commonly the first step in the analysis workflow, allowing a preliminary understanding of the data and its general patterns and behaviors, providing grounds for the analyst to formulate hypotheses, and directing the next steps in the analysis. Graphs are powerful tools to aid in this step, enabling the analyst to visualize the data, create new views and concepts, and communicate them to a larger audience.

There is a vast amount of literature on statistics about visualizing information. The classic book, *Envisioning Information*, by Edward Tufte, demonstrates beautiful and useful examples of how to present information in graphical form. In another book, The *Visual Display of Quantitative Information*, Tufte enumerates a few qualities that a graph that will be used for analysis and transmitting information, including statistics, should have:

- Show the data
- Avoid distorting what the data has to say
- Make large datasets coherent
- Serve a reasonably clear purpose—description, exploration, tabulation, or decoration

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Graphs must reveal information. We should think about creating graphs with these principles in mind when creating an analysis.

A graph should also be able to stand out on its own, outside the analysis. Let's say that you are writing an analysis report that becomes extensive. Now, we need to create a

summary of that extensive analysis. To make the analysis' points clear, a graph can be used to represent the data. This graph should be able to support the summary without the entire extensive analysis. To enable the graph to give more information and be able to stand out on its own in the summary, we have to add more information to it, such as a title and labels.

Exercise 8: Plotting an Analytical Function

In this exercise, we will create a basic plot using the Matplotlib libraries, where we will visualize a function of two variables, for example, **`y = f(x)`**, where **`f(x)`** is **`x^2`**:

1. First, create a new Jupyter notebook and import all the required libraries:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

2. Now, let's generate a dataset and plot it using the following code:

```
x = np.linspace(-50, 50, 100)
y = np.power(x, 2)
```

3. Use the following command to create a basic graph with Matplotlib:

```
plt.plot(x, y)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

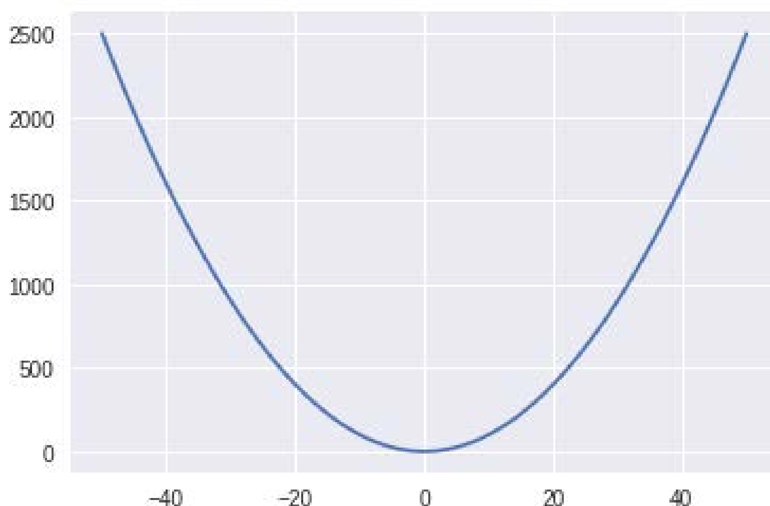


Figure 2.1: Basic plot of X and Y axis

4. Now, modify the data generation function from **`x^2`** to **`x^3`**, keeping the same interval of **`[-50,50]`** and recreate the line plot:

```
y_hat = np.power(x, 3)
plt.plot(x, y_hat)
```

The output is as follows:

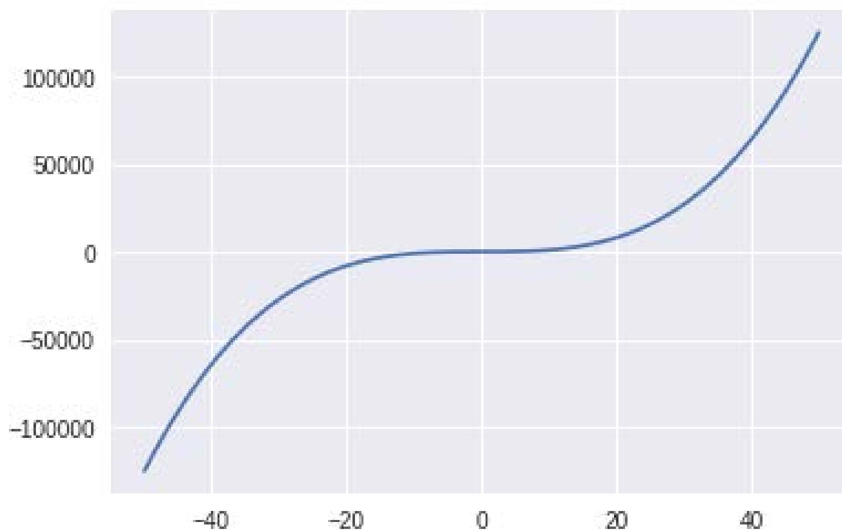


Figure 2.2: Basic plot of X and Y axis

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

As you can see, the shape of the function changed, as expected. The basic type of graph that we used was sufficient to see the change between the **`y`** and **`y_hat`** values. But some questions remain: we plotted only a mathematical function, but generally the data that we are collecting has dimensions, such as length, time, and mass. How can we add this information to the plot? How do we add a title? Let's explore this in the next section.

Components of a Graph

Each graph has a set of common components that can be adjusted. The names that Matplotlib uses for these components are demonstrated in the following graph:

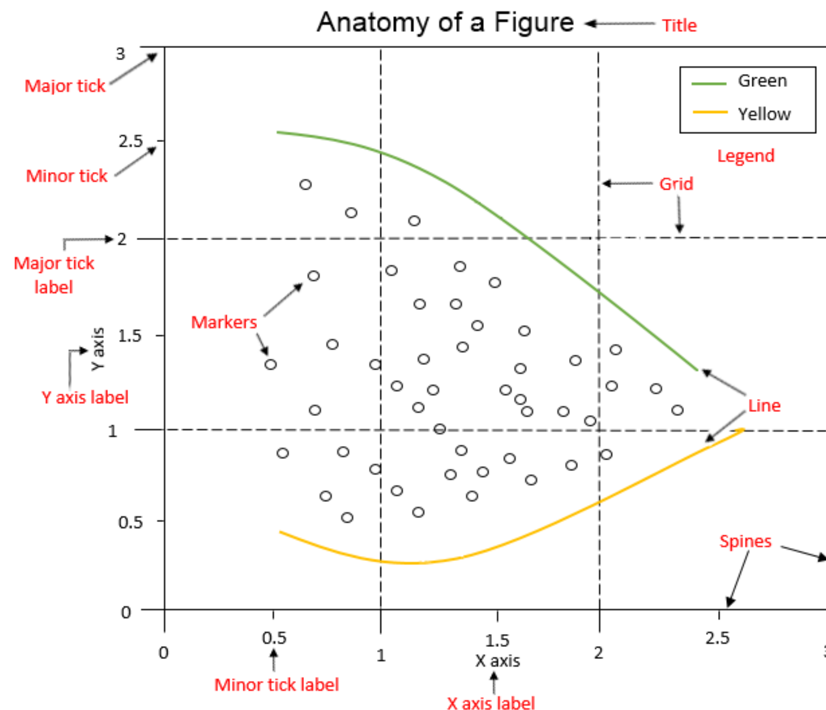


Figure 2.3: Components of a graph

The components of a graph are as follows:

- **Figure**: The base of the graph, where all the other components are drawn.
- **Axis**: Contains the figure elements and sets the coordinate system.
- **Title**: The title gives the graph its name.
- **X-axis label**: The name of the x-axis, usually named with the units.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

- **Y-axis label**: The name of the y-axis, usually named with the units.
- **Legend**: A description of the data plotted in the graph, allowing you to identify the curves and points in the graph.
- **Ticks and tick labels**: They indicate the points of reference on a scale for the graph, where the values of the data are. The labels indicate the values themselves.
- **Line plots**: These are the lines that are plotted with the data.
- **Markers**: Markers are the pictograms that mark the point data.
- **Spines**: The lines that delimit the area of the graph where data is plotted.

Each of these components can be configured to adapt to the needs of the visualization task at hand. We will go through each type of graph and how to adapt the components as previously described.

Exercise 9: Creating a Graph

There are several ways to create a graph with Matplotlib. The first one is closely related to the MATLAB way of doing it, called **Pyplot**. Pyplot is an API that is a *state-based interface for Matplotlib*, meaning that it keeps the configurations and other parameters in the object itself. Pyplot is intended as a simple case.

Perform the following steps to create the graph of a sine function using the Matplotlib library:

1. Import all the required libraries, as we did in the previous exercise:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

   A second API, called **object-oriented API**, is intended for more complex plots, allowing for more flexibility and configuration. The usual way to access this API is to create a **figure** and **axes** using the **`plt.subplots`** module.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Now, to get a figure and an axis, use the following command:

```
fig, ax = plt.subplots()
```

> **Note**
>
> The figure is the top container for all other graph components. The axes set things such as the axis, the coordinate system, and contain the plot elements, such as lines, text, and many more.

3. To add a plot to a graph created using the object-oriented API, use the following command:

```
x = np.linspace(0,100,500)
y = np.sin(2*np.pi*x/100)
ax.plot(x, y)
```

The output is as follows:

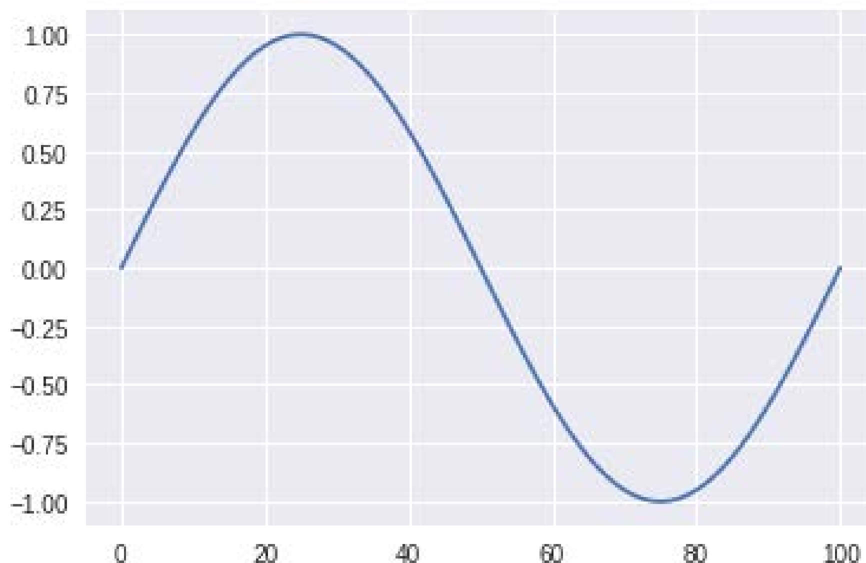


Figure 2.4: Plot output using the object-oriented API

We are adding a line plot to the **`ax`** axis that belongs to the **`fig`** figure. Changes to the graph, such as label names, title, and so on, will be demonstrated later in this chapter. For now, let's look at how to create each kind of graph that we may use in an analysis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 10: Creating a Graph for a Mathematical Function

In *Exercise 1: Plotting an Analytical Function*, we created a graph for a mathematical function using the MATLAB-like interface, Pyplot. Now that we know how to use the Matplotlib object-oriented API, let's create a new graph using it. Analysts have a lot of flexibility in creating graphs, whatever the data source, when using the object-oriented API.

Let's create a plot using the object-oriented API with the NumPy **`sine`** function on the interval **`[0,100]`**:

1. Create the data points for the **`x`**-axis:

```
import numpy as np
x = np.linspace(0,100,200)
y = np.sin(x)
```

2. Create the API interface for Matplotlib:

```
%matplotlib inline
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
```

3. Add the graph using the axis object, **`ax`**:

```
ax.plot(x, y)
```

The output is as follows:

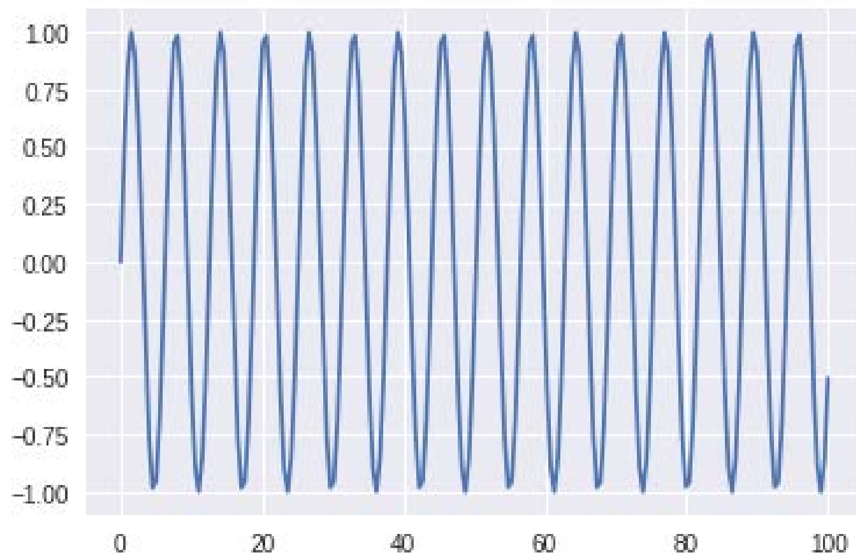


Figure 2.5: Graph for a mathematical function

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Notice that we again created a linear interval of values between `[0, 100]` using the `linspace` function, with `200` points. We then applied the `sine` function over these values to create the `y` axis. This is a common approach when creating data intervals.

Seaborn

Seaborn (https://seaborn.pydata.org/) is part of the PyData family of tools and is a visualization library based on Matplotlib with the goal of creating statistical graphs more easily. It can operate directly on DataFrames and series, doing aggregations and mapping internally. Seaborn uses color palettes and styles to make visualizations consistent and more informative. It also has functions that can calculate some statistics,

such as regression, estimation, and errors. Some specialized plots, such as violin plots and multi-facet plots, are also easy to create with Seaborn.

Which Tool Should Be Used?

Seaborn tries to make the creation of some common analysis graphs easier than using Matplotlib directly. Matplotlib can be considered more low-level than Seaborn, and although this makes it a bit more cumbersome and verbose, it gives analysts much more flexibility. Some graphs, which with Seaborn are created with one function call, would take several lines of code to achieve using Matplotlib.

There is no rule to determine whether an analyst should use only the pandas plotting interface, Matplotlib directly, or Seaborn. Analysts should keep in mind the visualization requirements and the level of configuration required to create the desired graph.

Pandas' plotting interface is easier to use but is more constrained and limited. Seaborn has several graph patterns ready to use, including common statistical graphs such as pair plots and boxplots, but requires that the data is formatted into a tidy format and is more opinionated on how the graphs should look. Matplotlib is the base for both cases and is more flexible than both, but it demands a lot more code to create the same visualizations as the two other options.

The rule of thumb that we use in this book is: how can I create the graph that I need with the least amount of code and without changing the data? With that in mind, we will use the three options, sometimes at the same time, to attain our visualization goals. Analysts should not be restricted to just one of the options. We encourage the use of any tool that creates a meaningful visualization.

Let's go through the most common kinds of graphs used in statistical analysis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Types of Graphs

The first type of graph that we will present is the **line graph** or **line chart**. A line graph displays data as a series of interconnected points on two axes (x and y), usually **Cartesian**, ordered commonly by the x-axis. Line charts are useful for demonstrating trends in data, such as in **time series**, for example.

A graph related to the line graph is the **scatter plot**. A scatter plot represents the data as points in Cartesian coordinates. Usually, two variables are demonstrated in this graph, although more information can be conveyed if the data is color-coded or size-coded by category, for example. Scatter plots are useful for showing the relationship and possible correlation between variables.

Histograms are useful for representing the distribution of data. Unlike the two previous examples, histograms show only one variable, usually on the x-axis, while the y-axis shows the frequency of occurrence of the data. The process of creating a histogram is a bit more involved than the line graph and the scatter plot, so we will explain them in a bit more detail.

Boxplots can also be used for representing frequency distributions, but it can help to compare groups of data using some statistical measurements, such as mean, median, and standard deviation. Boxplots are used to visualize the data distribution and outliers.

Each graph type has its applications and choosing the right type is paramount in the success of an analysis. For example, line graphs could be used to show trends in economic growth in the last century, while a boxplot for that would be hard to create. Another common data analysis task identifies correlations between variables: understanding whether two variables show related behaviors. A scatter plot is the tool commonly used to visualize this. Histograms are useful for visualizing data point numbers in a bin or a range, such as the number of cars whose efficiency is between 10 and 20 miles per gallon.

Line Graphs

Line graphs, as described in the previous section, connect the data points with a line. Line graphs are useful for demonstrating tendencies and trends. More than one line can be used on the same graph, for a comparison between the behavior of each line, although care must be taken so that the units on the graph are the same. They can also demonstrate the relationship between an independent and a dependent variable. A common case for this is time series.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Time Series Plots

Time series plots, as the name suggests, graphs the behavior of the data with respect to time. Time series graphs are used frequently in financial areas and environmental sciences. For instance, a historical series of temperature anomalies are shown in the following graph:

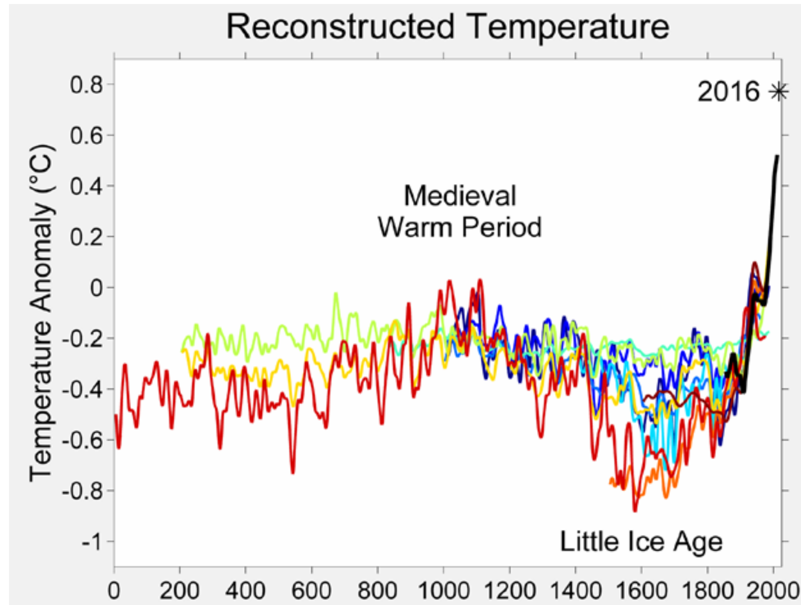


Figure 2.6: Time series plot

> **Source**
>
> https://upload.wikimedia.org/wikipedia/commons/c/c1/2000_Year_Temperature_Comparison.png

Usually, a time series graph has the `time` variable on the x-axis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 11: Creating Line Graphs Using Different Libraries

Let's compare the creation process between Matplotlib, Pandas, and Seaborn. We will create a Pandas DataFrame with random values and plot it using various methods:

1. Create a dataset with random values:

```
import numpy as np
X = np.arange(0,100)
Y = np.random.randint(0,200, size=X.shape[0])
```

2. Plot the data using the Matplotlib Pyplot interface:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(X, Y)
```

3. Now, let's create a Pandas DataFrame with the created values:

```
import pandas as pd
df = pd.DataFrame({'x':X, 'y_col':Y})
```

4. Plot it using the Pyplot interface, but with the **data** argument:

```
plt.plot('x', 'y_col', data=df)
```

The output is as follows:

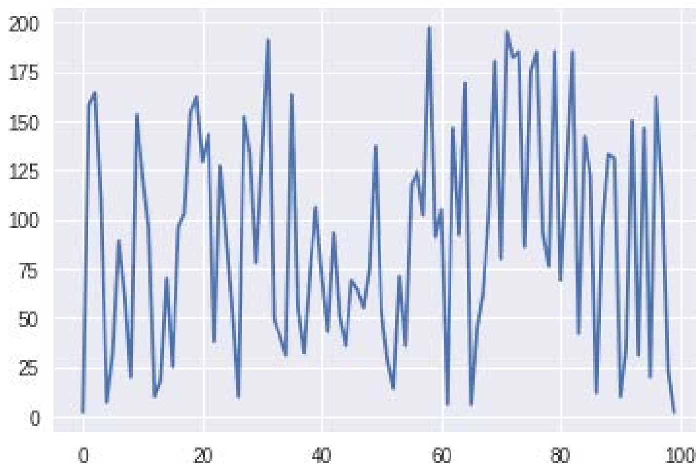


Figure 2.7: Line graphs using different libraries

5. With the same DataFrame, we can also plot directly from the Pandas DataFrame:

```
df.plot('x', 'y_col')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

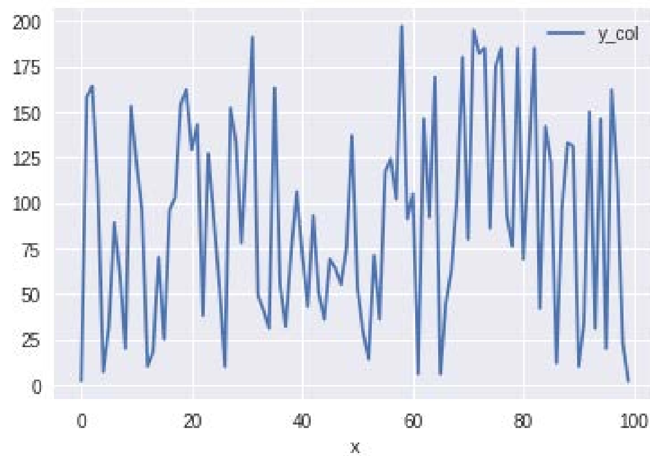


Figure 2.8: Line graphs from the pandas DataFrame

6. What about Seaborn? Let's create the same line plot with Seaborn:

```
import seaborn as sns
sns.lineplot(X, Y)
sns.lineplot('x', 'y_col', data=df)
```

The output is as follows:

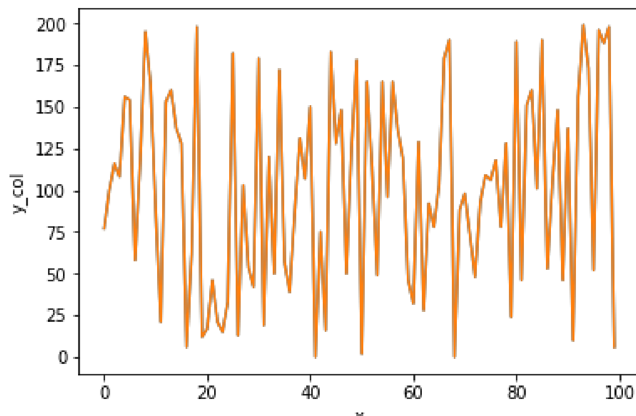


Figure 2.9: Line graphs from the Seaborn DataFrame

We can see that, in this case, the interface used by Matplotlib and Seaborn is quite similar.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Pandas DataFrames and Grouped Data

As we learned in the previous chapter, when analyzing data and using Pandas to do so, we can use the plot functions from Pandas or use Matplotlib directly. Pandas uses Matplotlib under the hood, so the integration is great. Depending on the situation, we can either plot directly from pandas or create a **figure** and an **axes** with Matplotlib and pass it to pandas to plot. For example, when doing a GroupBy, we can separate the data into a GroupBy key. But how can we plot the results of GroupBy? We have a few approaches at our disposal. We can, for example, use pandas directly, if the DataFrame is already in the right format:

> **Note**
>
> The following code is a sample and will not get executed.

```
fig, ax = plt.subplots()
df = pd.read_csv('data/dow_jones_index.data')
df[df.stock.isin(['MSFT', 'GE', 'PG'])].groupby('stock')['volume'].
plot(ax=ax)
```

Or we can just plot each GroupBy key on the same plot:

```
fig, ax = plt.subplots()
df.groupby('stock').volume.plot(ax=ax)
```

For the following activity, we will use what we've learned in the previous chapter and read a CSV file from a URL and parse it. The dataset is the Auto-MPG dataset (https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson02/Dataset/auto-mpg.data).

> **Note**
>
> This dataset is a modified version of the dataset provided in the **StatLib** library. The original dataset is available in the **auto-mpg.data-original** file.
>
> The data concerns city-cycle fuel consumption in miles per gallon, in terms of three multivalued discrete and five continuous attributes.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 4: Line Graphs with the Object-Oriented API and Pandas DataFrames

In this activity, we will create a time series line graph from the Auto-MPG dataset as a

first example of plotting using pandas and the object-oriented API. This kind of graph is common in analysis and helps to answer questions such as "is the average horsepower increasing or decreasing with time?"

Now, follow these procedures to plot a graph of average horsepower per year using pandas and while using the object-oriented API:

1. Import the required libraries and packages into the Jupyter notebook.
2. Read the Auto-MPG dataset into the Spark object.
3. Provide the column names to simplify the dataset, as illustrated here:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

4. Now read the new dataset with column names and display it.
5. Convert the **horsepower** and **year** data types to float and integer.
6. Now plot the graph of average horsepower per year using pandas:

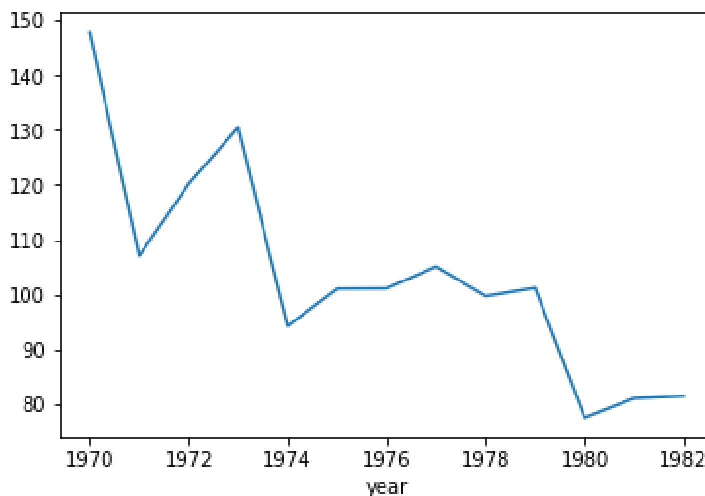


Figure 2.10: Line Graphs with the Object-Oriented API and Pandas DataFrame

> **Note**
>
> The solution for this activity can be found on page 205.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that we are using the plot functions from pandas but passing the axis that we created directly with Matplotlib as an argument. As we saw in the previous chapter, this is not required, but it will allow you to configure the plot outside pandas and change its configurations later. This same behavior can be applied to the other kinds of graphs. Let's now work with scatter plots.

Scatter Plots

To understand the correlation between two variables, scatter plots are generally used because they allow the distribution of points to be seen. Creating a scatter plot with Matplotlib is similar to creating a line plot, but instead of using the **`plot`** method, we use the **`scatter`** method.

Let's look at an example using the Auto-MPG dataset (https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/):

```
fig, ax = plt.subplots()
ax.scatter(x = df['horsepower'], y=df['weight'])
```

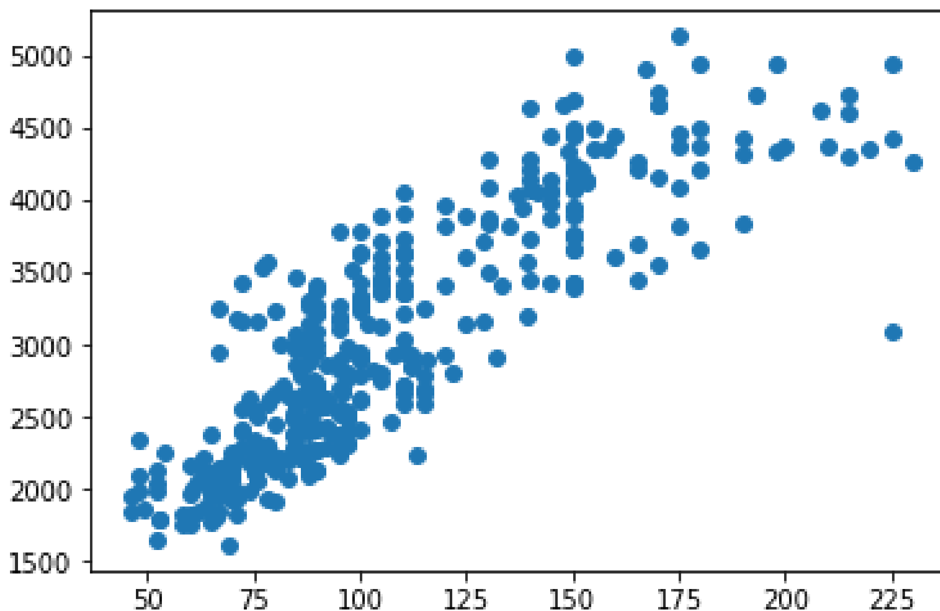


Figure 2.11: Scatter plot using Matplotlib library

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Note that we called the scatter method directly from the axis. In Matplotlib parlance, we added a scatter plot to the axis, **ax**, that belongs to the **fig** figure. We can also add more dimensions to the plot, such as the **color** and point **size**, easily with Seaborn:

```
import seaborn as sns

sns.scatterplot(data=df, x='horsepower', y='weight', hue='cylinders',
size='mpg')
```

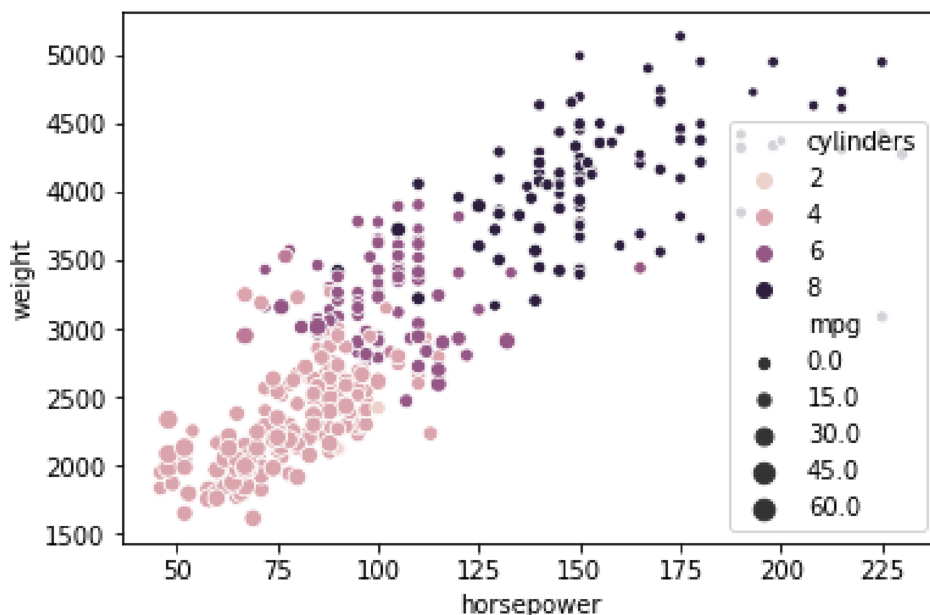


Figure 2.12: Scatter plot using Seaborn library

As we can see, scatter plots are quite helpful for understanding the relationship between two variables, or even more. We can infer, for example, that there is a positive correlation between **horsepower** and **weight**. We can also easily see an outlier with

the scatter plot, which could be more complicated when working with other kinds of graphs. The same principles for grouped data and pandas DataFrames that we saw on the line graphs apply here for the scatter plot.

We can generate a scatter plot directly from pandas using the **kind** parameter:

```
df.plot(kind='scatter', x='horsepower', y='weight')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Create a figure and pass it to Pandas:

```
fig, ax = plt.subplots()
df.plot(kind='scatter', x='horsepower', y='weight', ax =ax)
```

Activity 5: Understanding Relationships of Variables Using Scatter Plots

To continue our data analysis and learn how to plot data, let's look at a situation where a scatter plot can help. For example, let's use a scatter plot to answer the following question:

Is there a relationship between horsepower and weight?

To answer this question, we need to create a scatter plot with the data from Auto-MPG:

1. Use the Auto-MPG dataset, already ingested.

> **Note**
>
> Please refer to the previous exercise for how to ingest the dataset.

2. Use the object-oriented API for Matplotlib:

```
%matplotlib inline
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
```

3. Create a scatter plot using the **`scatter`** method:

```
ax.scatter(x = df['horsepower'], y=df['weight'])
```

> **Note**
>
> The solution for this activity can be found on page 208.

We can identify a roughly linear relationship between horsepower and weight, with some outliers with higher horsepower and lower weight. This is the kind of graph that would help an analyst interpret the data's behavior.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Histograms

Histograms are a bit different from the graphs that we've seen so far, as they only try to visualize the distribution of one variable, instead of two or more. Histograms have the goal of visualizing the probability distribution of one variable, or in other words, counting the number of occurrences of certain values divided into fixed intervals, or bins.

The bins are consecutive and adjacent but don't need to have the same size, although this is the most common arrangement.

The choice of the number of bins and bin size is more dependent on the data and the analysis goal than any fixed, general rule. The larger the number of bins, the smaller (narrower) the size of each bin, and vice versa. When data has a lot of noise or variation, for example, a small number of bins (with a large bin) will show the general outline of the data, reducing the impact of the noise in a first analysis. A larger number of bins is more useful when the data has a higher density.

Exercise 12: Creating a Histogram of Horsepower Distribution

As we strive to understand the data, we now want to see the horsepower distribution over all cars. Analysis questions with an adequate histogram are, for example: what is the most frequent value of a variable? Is the distribution centered or does it have a tail? Let's plot a histogram of the horsepower distribution:

1. Import the required libraries into the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

    ```
    import pandas as pd
    import numpy as np
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    import seaborn as sns
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
    auto-mpg.data"
    df = pd.read_csv(url)
    ```

2. Provide the column names to simplify the dataset as illustrated here:

    ```
    column_names = ['mpg', 'Cylinders', 'displacement', 'horsepower',
    'weight', 'acceleration', 'year', 'origin', 'name']
    ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
df.head()
```

The plot is as follows:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Figure 2.13: The auto-mpg dataset

4. Convert the **horsepower** and **year** data types to float and integer using the following command:

```
df.loc[df.horsepower == '?', 'horsepower'] = np.nan
df['horsepower'] = pd.to_numeric(df['horsepower'])
df['full_date'] = pd.to_datetime(df.year, format='%y')
df['year'] = df['full_date'].dt.year
```

5. Create a graph directly from the Pandas DataFrame using the **plot** function and **kind='hist'**:

```
df.horsepower.plot(kind='hist')
```

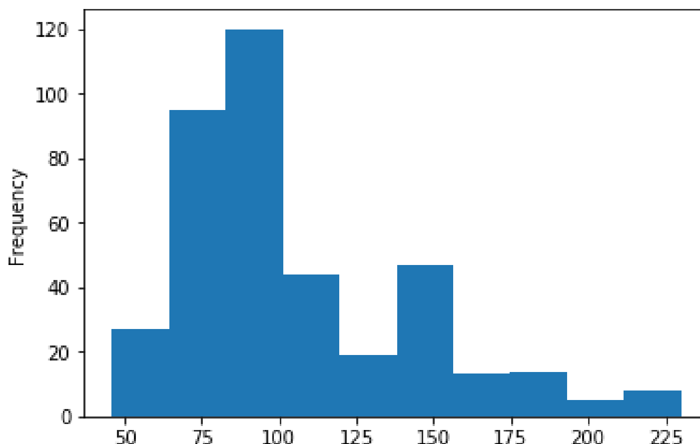


Figure 2.14: Histogram plot

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Identify the **horsepower** concentration:

```
sns.distplot(df['weight'])
```

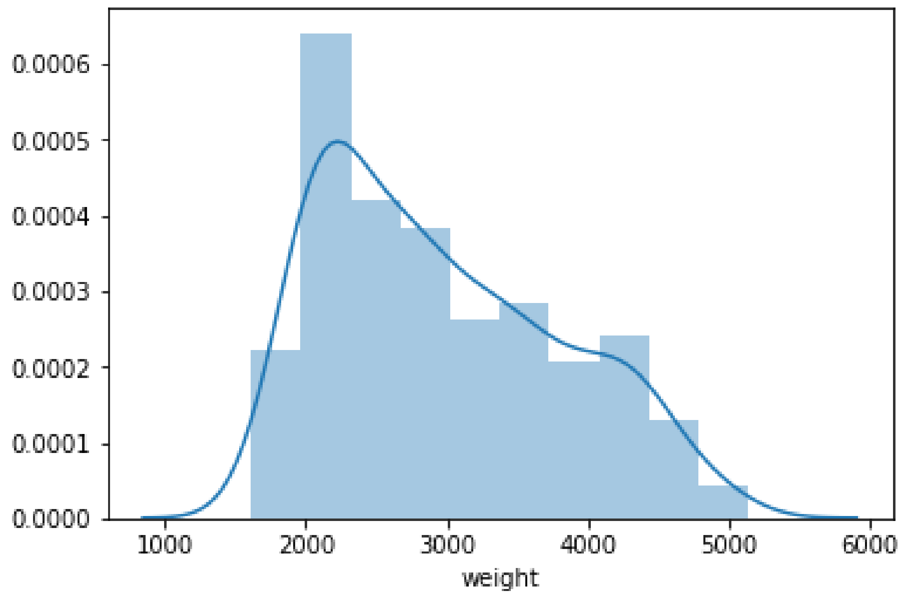


Figure 2.15: Histogram concentration plot

We can see in this graph that the value distribution is skewed to the left, with more cars with horsepower of between **50** and **100** than greater than **200**, for example. This could be quite useful in understanding how some data varies in an analysis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Boxplots

Boxplots are also used to see variations in values, but now within each column. We want to see how values compare when grouped by another variable, for example. Because of their format, boxplots are sometimes called **whisker plots** or **box** and **whisker plots** because of the lines that extend vertically from the main box:

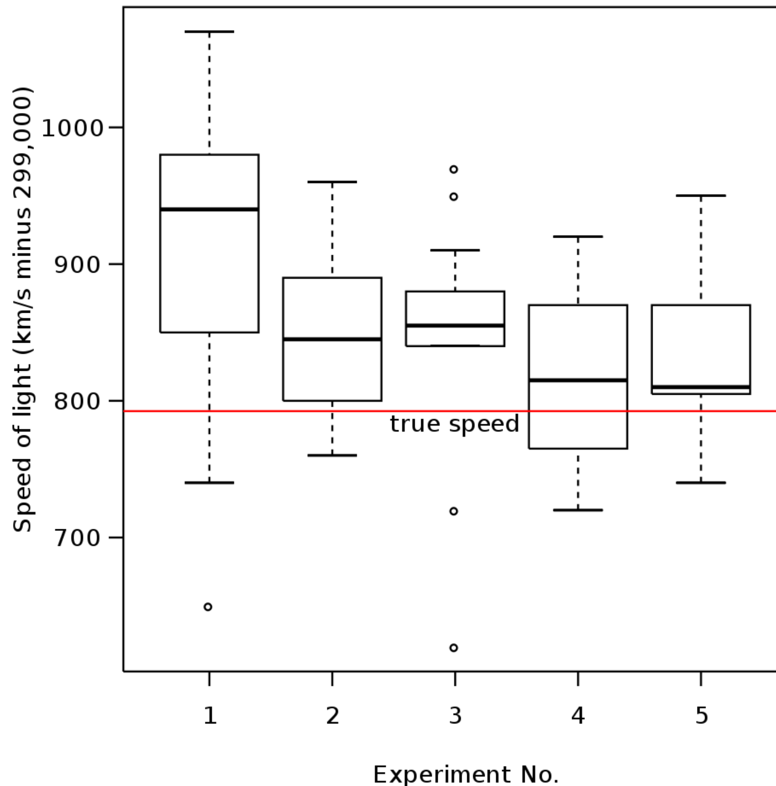


Figure 2.16: Boxplot

> **Source**
>
> https://en.wikipedia.org/wiki/File:Michelsonmorley-boxplot.svg

A boxplot uses quartiles (first and third) to create the boxes and whiskers. The line in the middle of the box is the second quartile – the median. The whiskers definition can vary, such as using one standard deviation above and below the mean of the data, but it's common to use 1.5 times the interquartile range (Q3 – Q1) from the edges of the box. Anything that passes these values, either above or below, is plotted as a `dot` and is usually considered an outlier.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 13: Analyzing the Behavior of the Number of Cylinders and Horsepower Using a Boxplot

Sometimes we want not only to see the distribution of each variable, but also to see the variation of the variable of interest with respect to another attribute. We would like to know, for instance, how the horsepower varies given the number of cylinders. Let's create a boxplot with Seaborn, comparing the horsepower distribution to the number of cylinders:

1. Import the required libraries into the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data"
df = pd.read_csv(url)
```

2. Provide the column names to simplify the dataset, as illustrated here:

```
column_names = ['mpg', 'Cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
df.head()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The plot is as follows:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Figure 2.17: The auto-mpg dataset

4. Convert the data type of horsepower and year to float and integer using the following command:

```
df.loc[df.horsepower == '?', 'horsepower'] = np.nan
df['horsepower'] = pd.to_numeric(df['horsepower'])
df['full_date'] = pd.to_datetime(df.year, format='%y')
df['year'] = df['full_date'].dt.year
```

5. Create a boxplot using the Seaborn **boxplot** function:

```
sns.boxplot(data=df, x="cylinders", y="horsepower")
```

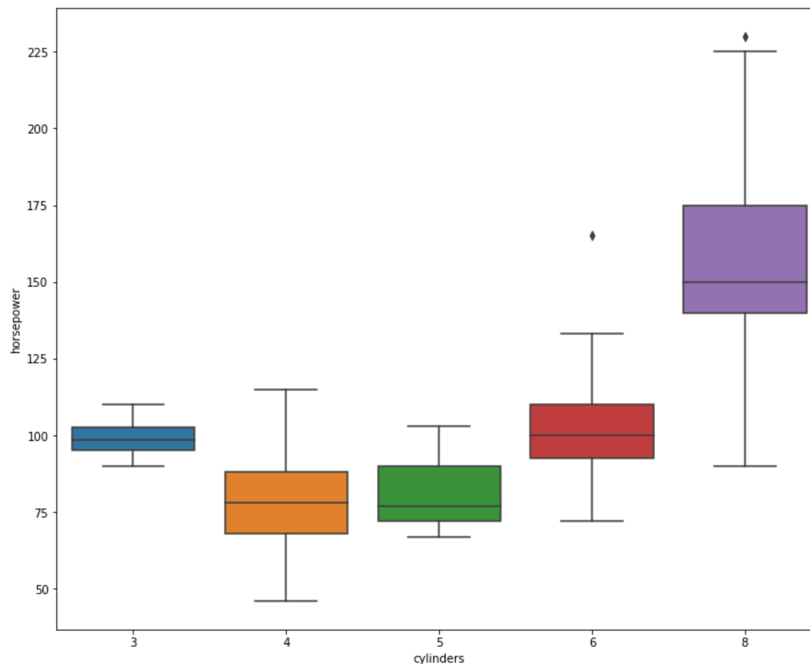


Figure 2.18: Boxplot using the Seaborn boxplot function

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Now, just for comparison purposes, create the same **boxplot** using pandas directly:

```
df.boxplot(column='horsepower', by='cylinders')
```

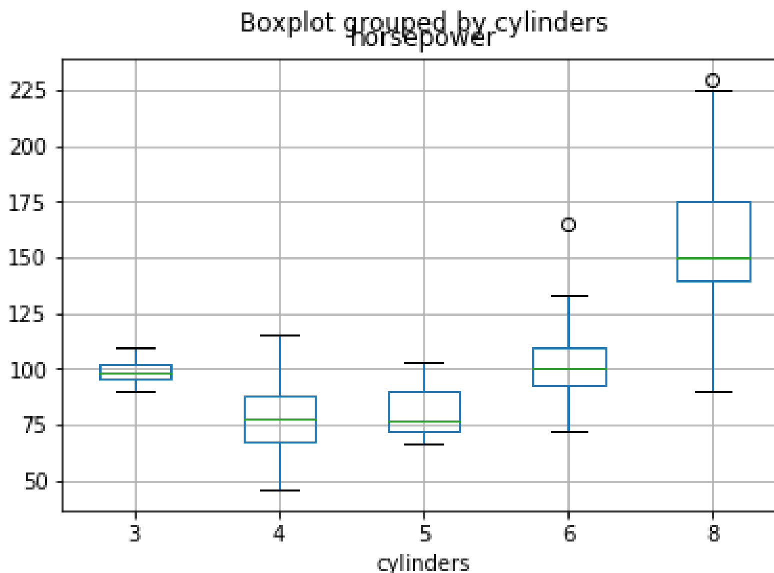


Figure 2.19: Boxplot using pandas

On the analysis side, we can see that the variation range from **3** cylinders is smaller than for **8** cylinders for horsepower. We can also see that **6** and **8** cylinders have outliers in the data. As for the plotting, the Seaborn function is more complete, showing different colors automatically for different numbers of cylinders, and including the name of the DataFrame columns as labels in the graph.

Changing Plot Design: Modifying Graph Components

So far, we've looked at the main graphs used in analyzing data, either directly or grouped, for comparison and trend visualization. But one thing that we can see is that the design of each graph is different from the others, and we don't have basic things such as a title and legends.

We've learned that a graph is composed of several components, such as a graph **title**, **x** and **y** labels, and so on. When using Seaborn, the graphs already have **x** and **y** labels, with the names of the columns. With Matplotlib, we don't have this. These changes are not only cosmetic.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The understanding of a graph can be greatly improved when we adjust things such as line width, color, and point size too, besides labels and titles. A graph must be able to stand on its own, so title, legends, and units are paramount. How can we apply the concepts that we described previously to make good, informative graphs on Matplotlib and Seaborn?

The possible number of ways that plots can be configured is enormous. Matplotlib is powerful when it comes to configuration, but at the expense of simplicity. It can be cumbersome to change some basic parameters in a graph using Matplotlib and this is where Seaborn and other libraries can be of help. But in some cases, this is desirable, for instance in a custom graph, so having this capacity somewhere in the stack is necessary. We will focus on how to change a few basic plot parameters in this section.

Title and Label Configuration for Axis Objects

As we said before, the object-oriented API for Matplotlib is the one that provides greater flexibility. Let's explore how to configure the title and labels on axis objects in the following exercise.

Exercise 14: Configuring a Title and Labels for Axis Objects

Perform the following steps to configure a title and labels for the axis objects. We will continue from the previous exercise and follow these steps:

1. Set the **title**, the x-axis label, and the y-axis label by calling the **set** method:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.set(title="Graph title", xlabel="Label of x axis (units)",
ylabel="Label of y axis (units)")
ax.plot()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The plot is as follows:

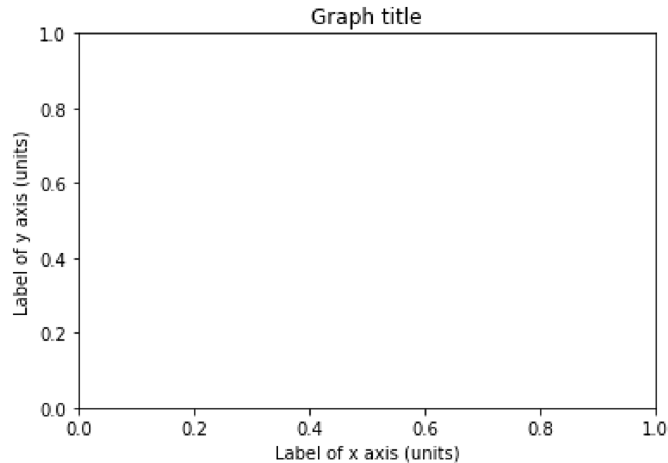


Figure 2.20: Configuring a title and labels

2. The legends for the plot can be either passed externally, when using only Matplotlib, or can be set on the Pandas plot and plotted with the axes. Use the following command to plot the legends:

```
fig, ax = plt.subplots()
df.groupby('year')['horsepower'].mean().plot(ax=ax, label='horsepower')
ax.legend()
```

The plot is as follows:

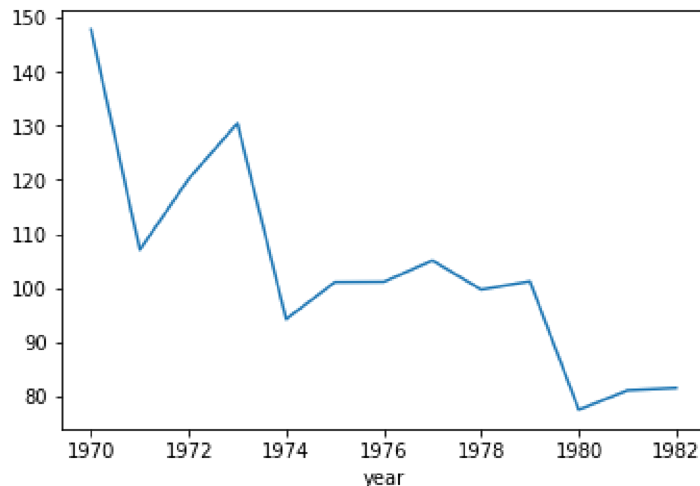


Figure 2.21: Line graph with legends

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. The alternative method for plotting the legends is as follows:

```
fig, ax = plt.subplots()
df.groupby('year')['horsepower'].mean().plot(ax=ax)
ax.legend(['horsepower'])
```

The plot is as follows:

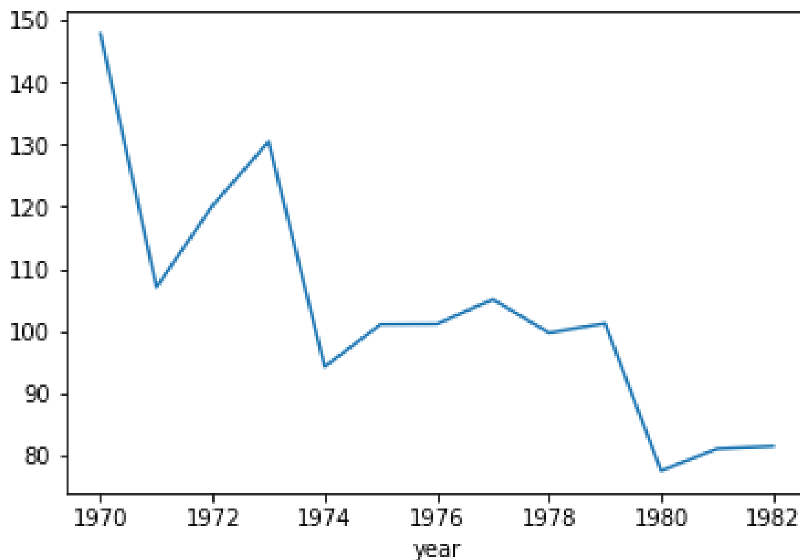


Figure 2.22: Line graph with legends (alternate method)

Line Styles and Color

For line graphs, the color, the weight, markers, and the style of the lines can be configured with the **`ls`**, **`lw`**, **`marker`**, and **`color`** parameters:

```
df.groupby('year')['horsepower'].mean().plot(ls='-.', color='r', lw=3)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

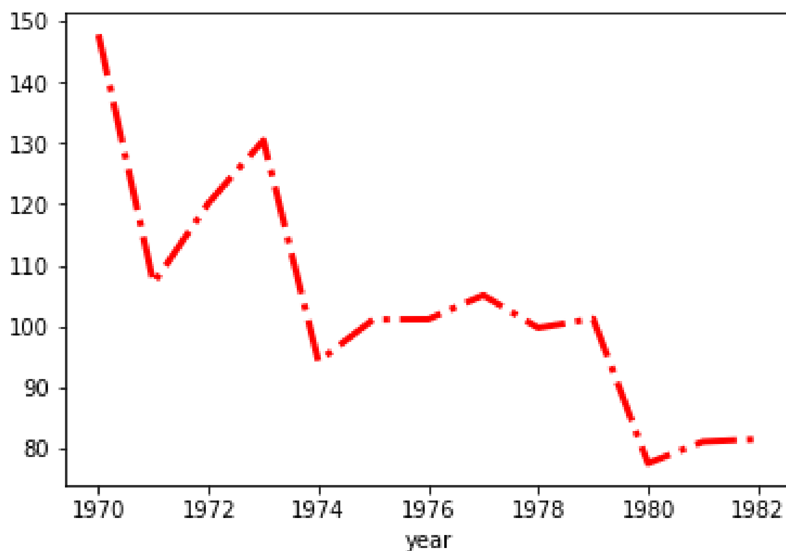


Figure 2.23: Line graph with color and style

Figure Size

We can also configure the size of the figure. The **`figsize`** parameter can be passed to all plot functions as a tuple (**`x`**-axis, **`y`**-axis) with the size in inches:

```
df.plot(kind='scatter', x='weight', y='horsepower', figsize=(20,10))
```

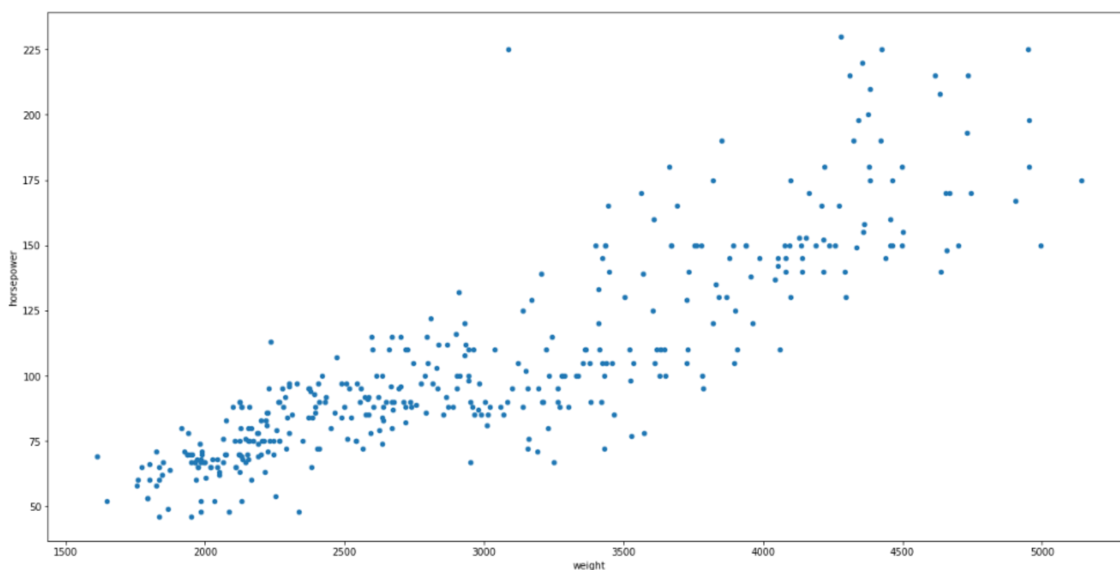


Figure 2.24: Plot with bigger figure size

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 15: Working with Matplotlib Style Sheets

Matplotlib has some style sheets that define general rules for graphs, such as **background color**, **ticks**, **graph colors**, and **palettes**. Let's say that we want to change the style so our graph has better colors for printing. To accomplish that, follow these steps:

1. Let's first print the list of available styles using the following command:

   ```
   import matplotlib.pyplot as plt
   print(plt.style.available)
   ```

   The output is as follows:

   ```
   ['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight',
   'ggplot', 'grayscale', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-
   dark-palette', 'seaborn-dark', 'seaborn-darkgrid', 'seaborn-deep',
   'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel',
   'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white',
   'seaborn-whitegrid', 'seaborn', 'Solarize_Light2', 'tableau-colorblind10',
   '_classic_test']
   ```

2. Now, let's create a scatter plot with the style as **`classic`**. Make sure you import the Matplotlib library first before proceeding:

   ```
   %matplotlib inline

   import numpy as np
   import matplotlib.pyplot as plt
   url = ('https://raw.githubusercontent.com/TrainingByPackt/Big-Data-
   Analysis-with-Python/master/Lesson02/Dataset/auto-mpg.data')
   df = pd.read_csv(url)
   column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
   'weight', 'acceleration', 'year', 'origin', 'name']
   df = pd.read_csv(url, names= column_names, delim_whitespace=True)

   df.loc[df.horsepower == '?', 'horsepower'] = np.nan
   df['horsepower'] = pd.to_numeric(df['horsepower'])
   plt.style.use(['classic'])
   df.plot(kind='scatter', x='weight', y='horsepower')
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

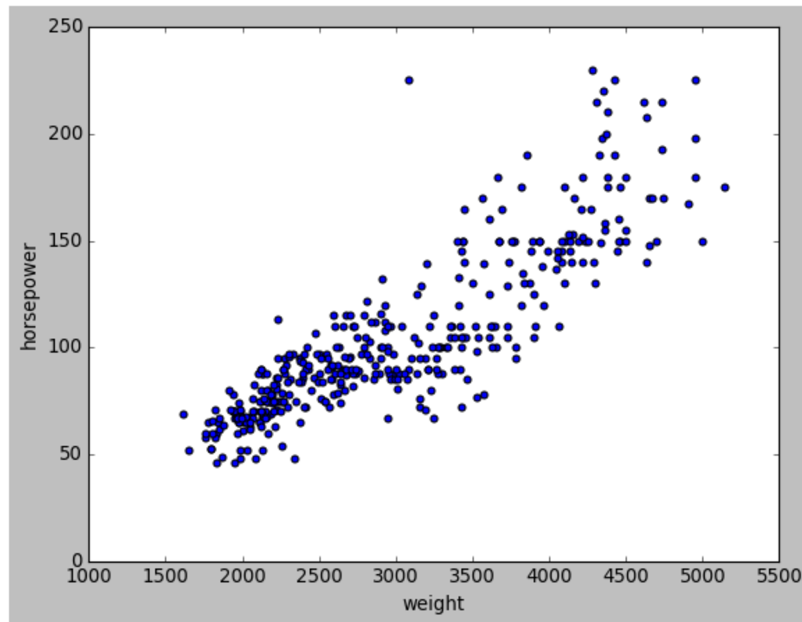


Figure 2.25: Scatter plot with the style as classic

> **Note**
>
> To use a style sheet, use the following command:
>
> ```
> plt.style.use('presentation')
> ```

One of the changes that Seaborn makes when it is imported is to add some styles to the list of available ones. Styles are also useful when creating images for different audiences, such as one for visualization in a notebook and another for printing or displaying in a presentation.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exporting Graphs

After generating our visualizations and configuring the details, we can export our graphs to a hard copy format, such as PNG, JPEG, or SVG. If we are using the interactive API in the notebook, we can just call the **savefig** function over the **pyplot** interface, and the last generated graph will be exported to the file:

```
df.plot(kind='scatter', x='weight', y='horsepower', figsize=(20,10))
plt.savefig('horsepower_weight_scatter.png')
```

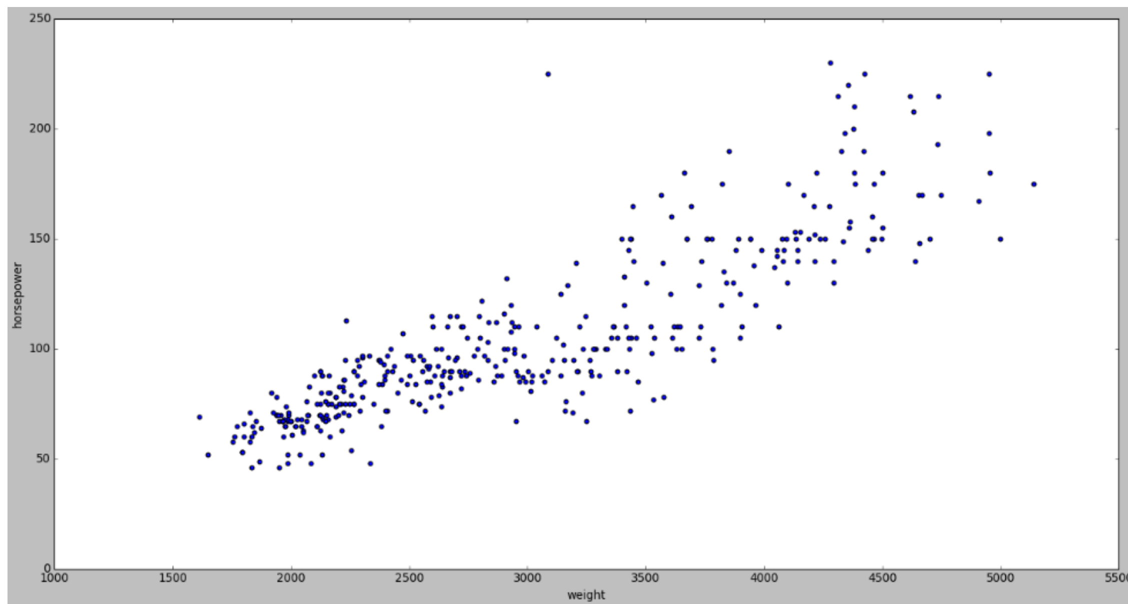


Figure 2.26: Exporting the graphs

All plot configurations will be carried to the **plot**. To export a graph when using the object-oriented API, we can call **savefig** from the figure:

```
fig, ax = plt.subplots()
df.plot(kind='scatter', x='weight', y='horsepower', figsize=(20,10), ax=ax)
fig.savefig('horsepower_weight_scatter.jpg')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

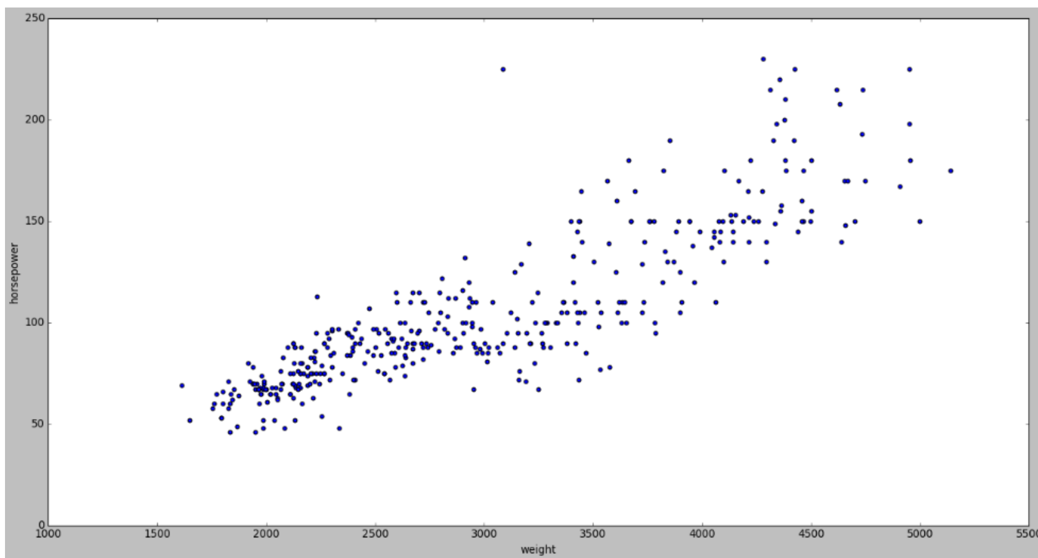


Figure 2.27: Saving the graph

We can change some parameters for the saved image:

- **dpi**: Adjust the saved image resolution.
- **facecolor**: The face color of the figure.
- **edgecolor**: The edge color of the figure, around the graph.
- **format**: Usually PNG, PDF, PS, EPS, JPG, or SVG. Inferred from the filename extension.

Seaborn also uses the same underlying Matplotlib mechanism to save figures. Call the **savefig** method directly from a Seaborn graph:

```
sns_scatter = sns.scatterplot(data=df, x='horsepower', y='weight',
hue='cylinders', size='mpg')

plt.savefig('scatter_fig.png', dpi=300)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

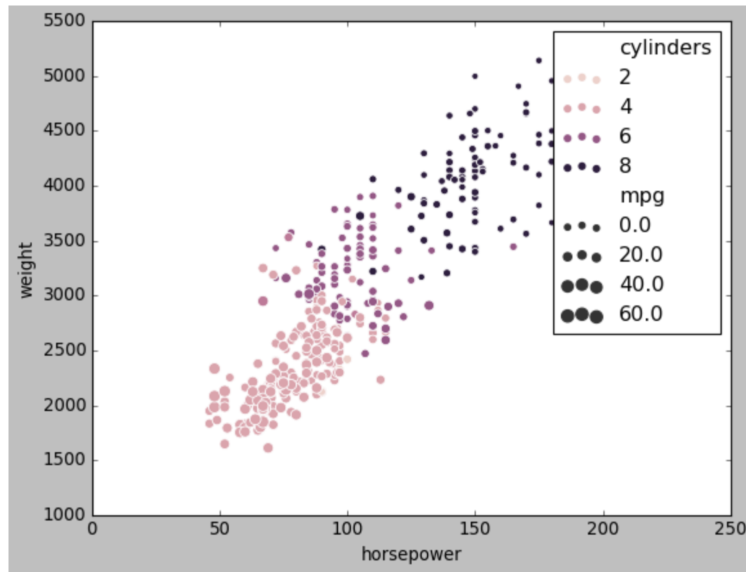


Figure 2.28: Plot using the savefig method

With these complementary options, the analyst has the capability of generating visualizations for different audiences, either in notebooks, on websites, or in print.

Activity 6: Exporting a Graph to a File on Disk

Saving our work to a file is a good way to enable sharing the results in different media. It also helps if we want to keep it for future reference. Let's create a graph and save it to disk:

1. Ingest the Auto-MPG dataset.
2. Create any kind of graph using the Matplotlib object-oriented API. For example, here's a histogram on weight:

```
%matplotlib inline
import matplotlib.pyplot
fig, ax = plt.subplots()
df.weight.plot(kind='hist', ax=ax)
```

3. Export this to a PNG file using the **savefig** function:

```
fig.savefig('weight_hist.png')
```

> **Note**
>
> The solution for this activity can be found on page 209.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 7: Complete Plot Design

To make our graphs stand alone, separated from analysis, we need to add more information to it, for other analysis or for other users be able to grasp the graph content and understand what's being represented. We will now combine all of what we've learned in this chapter to create a complete graph, including a title, labels, and legends, and adjust the plot size.

As an analyst, we want to understand whether the average miles per year increased or not, and we want to group by number of cylinders. For example, what is the behavior, in fuel consumption, of a car with three cylinders, over time? Is it higher or lower than a car with four cylinders?

Follow these steps to create our final graph:

1. Ingest the Auto-MPG dataset.
2. Perform the **`groupby`** operations on **`year`** and **`cylinders`**, and unset the option to use them as indexes.
3. Calculate the average miles per gallon over the grouping and set year as index.
4. Set year as the DataFrame index.
5. Create the figure and axes using the object-oriented API.
6. Perform the **`groupby`** operations on the **`df_g`** dataset by cylinders and plot the miles per gallon variable using the axes created with size **`(10,8)`**.
7. Set the **title**, **`x`** label, and **`y`** label on the axes.
8. Include the legends in the plot.
9. Save the figure to disk as a PNG file.

> **Note**
>
> The solution for this activity can be found on page 211.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

We can infer from this graph that cars with four cylinders are more economical than cars with eight cylinders. We can also infer that all cars increased fuel efficiency during the studied period, with a decrease of four cylinders between 1980 and 1982.

> **Note**
>
> Notice that with the label axes and legend, the complicated transformation that was done using Pandas (group by and averaging, then setting the index) is easy to explain in the final result.

Summary

In this chapter, we have seen the importance of creating meaningful and interesting visualizations when analyzing data. A good data visualization can immensely help the analyst's job, representing data in a way that can reach larger audiences and explain concepts that could be hard to translate into words or to represent with tables.

A graph, to be effective as a data visualization tool, must show the data, avoid distortions, make understanding large datasets easy, and have a clear purpose, such as description or exploration. The main goal of a graph is to communicate data, so the analyst must keep that in mind when creating a graph. A useful graph is more desirable than a beautiful one.

We demonstrated some kinds of graphs commonly used in analysis: the line graph, the scatter plot, the histogram, and the boxplot. Each graph has its purpose and application, depending on the data and the goal. We have also shown how to create graphs directly from Matplotlib, from pandas, or a combination of both, with Matplotlib's APIs: Pyplot, the interactive API, and the object-oriented API. We finished this chapter with an explanation of the options for changing the appearance of a graph, from line styles to markers and colors, and how to save a graph to a hard copy format for printing or sharing.

There are plenty of ways to configure graphs that we didn't cover here. Visualization is a large field and the tools also have a large set of options.

In the next chapters, we will focus on data processing, including the manipulation of large-scale data with Hadoop and Spark. After going through the basics of both of those tools, we will return to the analysis process, which will include graphs in diverse forms.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3

Working with Big Data Frameworks

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the HDFS and YARN Hadoop components
- Perform file operations with HDFS
- Compare a pandas DataFrame with a Spark DataFrame
- Read files from a local filesystem and HDFS using Spark
- Write files in Parquet format using Spark
- Write partitioned files in Parquet for fast analysis
- Manipulate non-structured data with Spark

In this chapter, we will explore big data tools such as Hadoop and Spark.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

We saw in the previous chapters how to work with data using pandas and Matplotlib for visualization and the other tools in the Python data science stack. So far, the datasets that we have used have been relatively small and with a relatively simple structure. Real-life datasets can be orders of magnitude larger than can fit into the memory of a single machine, the time to process these datasets can be long, and the usual software tools may not be up to the task. This is the usual definition of what big data is: an amount of data that does not fit into memory or cannot be processed or analyzed in a reasonable amount of time by common software methods. What is big data for some may not be big data for others, and this definition can vary depending on who you ask.

Big Data is also associated with the 3 V's (later extended to 4 V's):

- **Volume**: Big data, as the name suggests, is usually associated with very large volumes of data. What is large depends on the context: for one system, gigabytes can be large, while for another, we have to go to petabytes of data.
- **Variety**: Usually, big data is associated with different data formats and types, such as text, video, and audio. Data can be structured, like relational tables, or unstructured, like text and video.
- **Velocity**: The speed at which data is generated and stored is faster than other systems and produced more continuously. Streaming data can be generated by platforms such as telecommunications operators or online stores, or even Twitter.
- **Veracity**: This was added later and tries to show that knowing the data that is being used and its meaning is important in any analysis work. We need to check whether the data corresponds with what we expect the data to be, that the transformation process didn't change the data, and whether it reflects what was collected.

But one aspect that makes big data compelling is the analysis component: big data platforms are created to allow analysis and information extraction over these large datasets. This is where this chapter starts: we will learn how to manipulate, store, and analyze large datasets using two of the most common and versatile frameworks for big data: Hadoop and Spark.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Hadoop

Apache Hadoop is a set of software components created for the parallel storage and computation of large volumes of data. The main idea at the time of its inception was to use commonly available computers in a distributed fashion, with high resiliency against failure and distributed computation. With its success, more high-end computers started to be used on Hadoop clusters, although commodity hardware is still a common use case.

By parallel storage, we mean any system that stores and retrieves stored data in a parallel fashion, using several nodes interconnected by a network.

Hadoop is composed of the following:

- **Hadoop Common**: the basic common Hadoop items
- **Hadoop YARN**: a resource and job manager
- **Hadoop MapReduce**: a large-scale parallel processing engine
- **Hadoop Distributed File System** (**HDFS**): as the name suggests, HDFS is a file system that can be distributed over several machines, using local disks, to create a large storage pool:

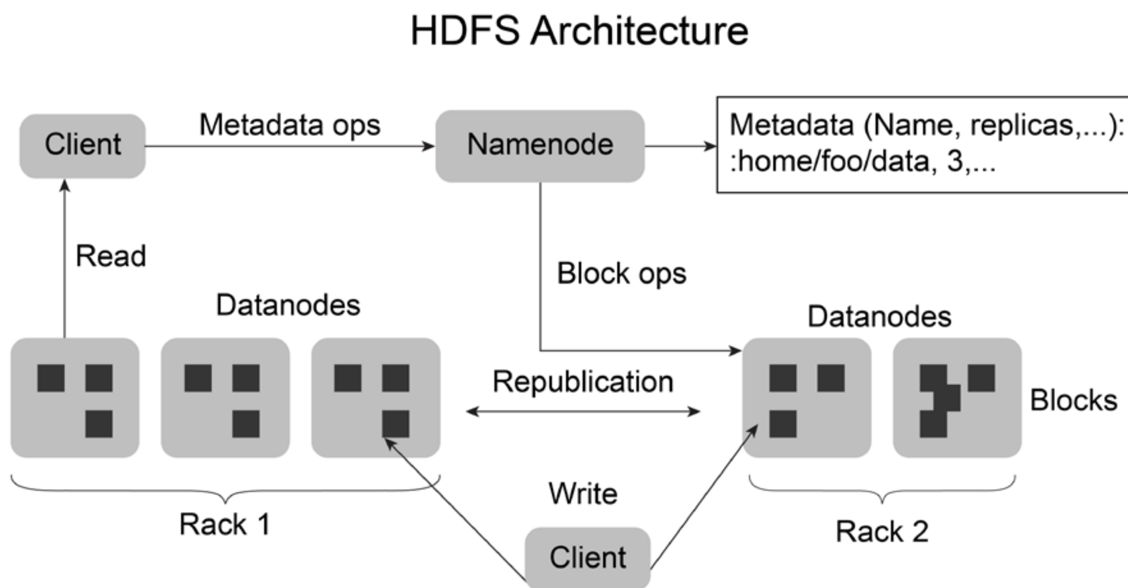


Figure 3.1: Architecture of HDFS

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Another important component is **YARN** (**Yet Another Resource Negotiator**), a resource manager and job scheduler for Hadoop. It is responsible for managing jobs submitted to the Hadoop cluster, allocating memory and CPU based on the required and available resources.

Hadoop popularized a parallel computation model called MapReduce, a distributed computation paradigm first developed by Google. It's possible to run a program using MapReduce in Hadoop directly. But since Hadoop's creation, other parallel computation paradigms and frameworks have been developed (such as Spark), so MapReduce is not commonly used for data analysis. Before diving into Spark, let's see how we can manipulate files on the HDFS.

Manipulating Data with the HDFS

The HDFS is a distributed file system with an important distinction: it was designed to run on thousands of computers that were not built specially for it—so-called **commodity hardware**. It doesn't require any special networking gear or special disks, it can run on common hardware. Another idea that permeates HDFS is that it is resilient: hardware will always fail, so instead of trying to prevent failure, the HDFS works around this by being extremely fault-tolerant. It assumes that failures will occur considering its scale, so the HDFS implements fault detection for fast and automatic recovery. It is also portable, running in diverse platforms, and can hold single files with terabytes of data.

One of the big advantages from a user perspective is that the HDFS supports traditional hierarchical file structure organization (folders and files in a tree structure), so users can create folders inside folders and files inside folders on each level, simplifying its use and operation. Files and folders can be moved around, deleted, and renamed, so users do not need to know about data replication or **NameNode/DataNode** architecture to use the HDFS; it would look similar to a Linux filesystem. Before demonstrating how to access files, we need to explain a bit about the addresses used to access Hadoop data. For example, the URI for accessing files in HDFS has the following format:

```
hdfs://hadoopnamenode.domainname/path/to/file
```

Where **`namenode.domainname`** is the address configured in Hadoop. Hadoop user guide (https://exitcondition.com/install-hadoop-windows/) details a bit more on how to access different parts of the Hadoop system. Let's look at a few examples to better understand how all this works.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 16: Manipulating Files in the HDFS

An analyst just received a large dataset to analyze and it's stored on an HDFS system. How would this analyst list, copy, rename, and move these files? Let's assume that the analyst received a file with raw data, named **new_data.csv**:

1. Let's start checking the current directories and files using the following command on the terminal if you are on Linux based system or command prompt if you are on Windows system:

```
hdfs dfs -ls /
```

2. We have a local file on disk, called **new_data.csv**, which we want to copy to the HDFS data folder:

```
hdfs dfs -put C:/Users/admin/Desktop/Lesson03/new_data.csv /
```

3. Notice that the last part of the command is the path inside HDFS. Now, create a folder in HDFS using the command **mkdir**:

```
hdfs dfs -mkdir /data
```

4. And move the file to a data folder in HDFS:

```
hdfs dfs -mv /data_file.csv /data
```

5. Change the name of the CSV file:

```
hdfs dfs -mv /data/new_data.csv /data/other_data.csv
```

6. Use the following command to check whether the file is present in the current location or not:

```
hadoop fs -ls /data
```

The output is as follows:

```
other_data.csv
```

> **Note**
>
> Commands after the HDFS part have the same name as commands in the Linux shell.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Knowing how to manipulate files and directories with the HDFS is an important part of big data analysis, but usually, direct manipulation is done only on ingestion. To analyze data, HDFS is not directly used, and tools such as Spark are more powerful. Let's see how to use Spark in sequence.

Spark

Spark (https://spark.apache.org) is a unified analytics engine for large-scale data processing. Spark started as a project by the University of California, Berkeley, in 2009, and moved to the Apache Software Foundation in 2013.

Spark was designed to tackle some problems with the Hadoop architecture when used for analysis, such as data streaming, SQL over files stored on HDFS and machine learning. It can distribute data over all computing nodes in a cluster in a way that decreases the latency of each computing step. Another Spark difference is its flexibility: there are interfaces for Java, Scala, SQL, R and Python, and libraries for different problems, such as MLlib for machine learning, GraphX for graph computation, and Spark Streaming, for streaming workloads.

Spark uses the worker abstraction, having a driver process that receives user input to start parallel executions, and worker processes that reside on the cluster nodes, executing tasks. It has a built-in cluster management tool and supports other tools, such as Hadoop YARN and Apache Mesos (and even Kubernetes), integrating into different environments and resource distribution scenarios.

Spark can also be very fast, because it first tries to distribute data over all nodes and keep it in memory instead of relying only on data on disk. It can handle datasets larger than the total available memory, shifting data between memory and disk, but making the process slower than if the entire dataset fitted in the total available memory of all nodes:

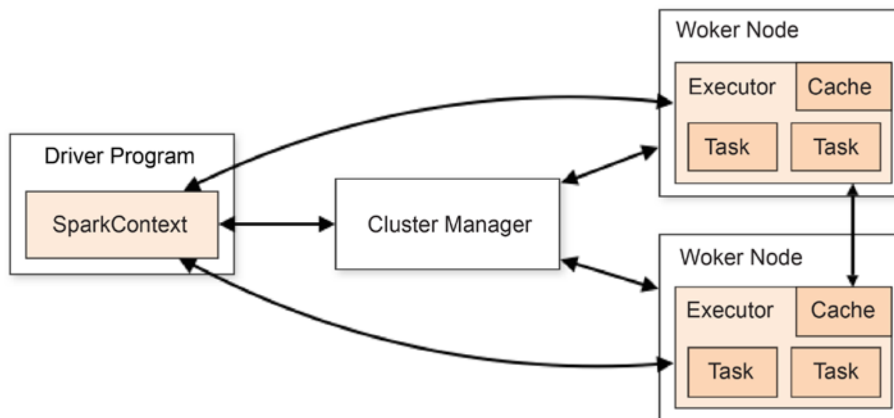


Figure 3.2: Working mechanism of Spark

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Other great advantages are that Spark has interfaces for a large variety of local and distributed storage systems, such as HDFS, Amazon S3, Cassandra, and others; can connect to RDBMS such as PostgreSQL and MySQL via JDBC or ODBC connectors; and can use the **Hive Metastore** to run SQL directly over a HDFS file. File formats such as CSV, Parquet, and ORC can also be read directly by Spark.

This flexibility can be a great help when working with big data sources, which can have varying formats.

Spark can be used either as an interactive shell with Scala, Python, and R, or as a job submission platform with the spark-submit command. The submit method is used to dispatch jobs to a Spark cluster coded in a script. The Spark shell interface for Python is called PySpark. It can be accessed directly from the terminal, where the Python version that is the default will be used; it can be accessed using the IPython shell or even inside a Jupyter notebook.

Spark SQL and Pandas DataFrames

The **RDD**, or **Resilient Distributed Dataset**, is the base abstraction that Spark uses to work with data. Starting on Spark version 2.0, the recommended API to manipulate data is the DataFrame API. The DataFrame API is built on top of the RDD API, although the RDD API can still be accessed.

Working with RDDs is considered low-level and all operations are available in the DataFrame API, but it doesn't hurt learning a bit more about the RDD API.

The SQL module enables users to query the data in Spark using SQL queries, similar to common relational databases. The DataFrame API is part of the SQL module, which works with structured data. This interface for data helps to create extra optimizations, with the same execution engine being used, independently of the API or language used to express such computations.

The DataFrame API is similar to the **Pandas DataFrame**. In Spark, a DataFrame is a distributed collection of data, organized into columns, with each column having a name. With Spark 2.0, the DataFrame is a part of the more general Dataset API, but as this API is only available for the Java and Scala languages, we will discuss only the DataFrame API (called **Untyped Dataset Operations** in the documentation).

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The interface for Spark DataFrames is similar to the pandas interface, but there are important differences:

- The first difference is that Spark DataFrames are **immutable**: after being created, they cannot be altered.
- The second difference is that Spark has two different kinds of operations: **transformations** and **actions**.

  Transformations are operations that are applied over the elements of a DataFrame and are queued to be executed later, not fetching data yet.

  Only when an **action** is called is data fetched and all queued transformations are executed. This is called lazy evaluation.

Exercise 17: Performing DataFrame Operations in Spark

Let's start using Spark to perform input/output and simple aggregation operations. The Spark interface, as we said before, was inspired by the pandas interface. What we learned in *Chapter 2, Statistical Visualizations Using Matplotlib and Seaborn*, can be applied here, making it easier to carry out more complex analysis faster, including aggregations, statistics, computation, and visualization on aggregated data later on. We want to read a CSV file, as we did before, to perform some analysis on it:

1. First, let's use the following command on Jupyter notebook to create a Spark session:

   ```
   from pyspark.sql import SparkSession
   >>> spark = SparkSession \
       .builder \
       .appName("Python Spark Session") \
       .getOrCreate()
   ```

2. Now, let's use the following command to read the data from the **`mydata.csv`** file:

   ```
   df = spark.read.csv('/data/mydata.csv', header=True)
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. As we said before, Spark evaluation is lazy, so if we want to show what values are inside the DataFrame, we need to call the action, as illustrated here:

```
df.show()
+------+----+-------+
|  name| age| height|
+------+----+-------+
|  Jonh|  22|   1.80|
|Hughes|  34|   1.96|
|  Mary|  27|   1.56|
+------+----+-------+
```

> **Note**
>
> This is not necessary with pandas: printing the DataFrame would work directly.

Exercise 18: Accessing Data with Spark

After reading our DataFrame and showing its contents, we want to start manipulating the data so that we can do an analysis. We can access data using the same NumPy selection syntax, providing the column name as **input**. Note that the object that is returned is of type **`Column`**:

1. Let's select one column from the DataFrame that we ingested in the previous exercise:

```
df['age'].Column['age']
```

This is different from what we've seen with pandas. The method that selects values from columns in a Spark DataFrame is **`select`**. So, let's see what happens when we use this method.

2. Using the same DataFrame again, use the **`select`** method to select the name column:

```
df.select(df['name'])DataFrame[age: string]
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now, it changed from **Column** to **DataFrame**. Because of that, we can use the methods for DataFrames. Use the **show** method for showing the results from the **select** method for **age**:

```
df.select(df['age']).show()
+---+
|age|
+---+
| 22|
| 34|
| 27|
+---+
```

4. Let's select more than one column. We can use the names of the columns to do this:

```
df.select(df['age'], df['height']).show()
+---+------+
|age|height|
+---+------+
| 22|  1.80|
| 34|  1.96|
| 27|  1.56|
+---+------+
```

This is extensible for other columns, selecting by name, with the same syntax. We will look at more complex operations, such as **aggregations with GroupBy**, in the next chapter.

Exercise 19: Reading Data from the Local Filesystem and the HDFS

As we saw before, to read files from the local disk, just give Spark the path to it. We can also read several other file formats, located in different storage systems. Spark can read files in the following formats:

- CSV
- JSON
- ORC
- Parquet
- Text

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

And can read from the following storage systems:

- JDBC
- ODBC
- Hive
- S3
- HDFS

Based on a URL scheme, as an exercise, let's read data from different places and formats:

1. Import the necessary libraries on the Jupyter notebook:

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark Session") \
    .getOrCreate()
```

2. Let's assume that we have to get some data from a JSON file, which is common for data collected from APIs on the web. To read a file directly from HDFS, use the following URL:

```
df = spark.read.json('hdfs://hadoopnamenode/data/myjsonfile.json')
```

Note that, with this kind of URL, we have to provide the full address of the HDFS endpoint. We could also use only the simplified path, assuming that Spark was configured with the right options.

3. Now, read the data into the Spark object using the following command:

```
df = spark.read.json('hdfs://data/myjsonfile.json')
```

4. So, we choose the format on the **read** method and the storage system on the access URL. The same method is used to access JDBC connections, but usually, we have to provide a user and a password to connect. Let's see how to connect to a PostgreSQL database:

```
url = "jdbc:postgresql://posgreserver:5432/mydatabase"
properties = {"user": "my_postgre_user",  password: "mypassword",
"driver": "org.postgresql.Driver"}
df = spark.read.jdbc(url, table = "mytable", properties = properties)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 20: Writing Data Back to the HDFS and PostgreSQL

As we saw with pandas, after performing some operations and transformations, let's say that we want to write the results back to the local file system. This can be very useful when we finish an analysis and want to share the results with other teams, or we want to show our data and results using other tools:

1. We can use the **`write`** method directly on the HDFS from the DataFrame:

```
df.write.csv('results.csv', header=True)
```

2. For the relational database, use the same URL and properties dictionary as illustrated here:

```
df = spark.write.jdbc(url, table = "mytable", properties = properties)
```

This gives Spark great flexibility in manipulating large datasets and combining them for analysis.

> **Note**
>
> Spark can be used as an intermediate tool to transform data, including aggregations or fixing data issues, and saving in a different format for other applications.

Writing Parquet Files

The Parquet data format (https://parquet.apache.org/) is binary, columnar storage that can be used by different tools, including Hadoop and Spark. It was built to support compression, to enable higher performance and storage use. Its column-oriented design helps with data selection for performance, as only the data in the required columns are retrieved, instead of searching for the data and discarding values in rows that are not required, reducing the retrieval time for big data scenarios, where the data is distributed and on disk. Parquet files can also be read and written by external applications, with a C++ library, and even directly from pandas.

The Parquet library is currently being developed with the **Arrow project** (https://arrow.apache.org/).

When considering more complex queries in Spark, storing the data in Parquet format can increase performance, especially when the queries need to search a massive dataset. Compression helps to decrease the data volume that needs to be communicated when an operation is being done in Spark, decreasing the network I/O. It also supports schemas and nested schemas, similar to JSON, and Spark can read the schema directly from the file.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Parquet writer in Spark has several options, such as mode (append, overwrite, ignore or error, the default option) and compression, a parameter to choose the compression algorithm. The available algorithms are as follows:

- **gzip**
- **lzo**
- **brottli**
- **lz4**
- Snappy
- Uncompressed

The default algorithm is **snappy**.

Exercise 21: Writing Parquet Files

Let's say that we received lots of CSV files and we need to do some analysis on them. We also need to reduce the data volume size. We can do that using Spark and Parquet.

Before starting our analysis, let's convert the CSV files to Parquet:

1. First, read the CSV files from the HDFS:

```
df = spark.read.csv('hdfs:/data/very_large_file.csv', header=True)
```

2. Write the CSV files in the DataFrame back to the HDFS, but now in Parquet format:

```
df.write.parquet('hdfs:/data/data_file', compression="snappy")
```

3. Now read the Parquet file to a new DataFrame:

```
df_pq = spark.read.parquet("hdfs:/data/data_file")
```

> **Note**
>
> The **write.parquet** method creates a folder named **data_file** with a file with a long name such as **part-00000-1932c1b2-e776-48c8-9c96-2875bf76769b-c000.snappy.parquet**.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Increasing Analysis Performance with Parquet and Partitions

An important concept that Parquet supports and that can also increase the performance of queries is partitioning. The idea behind partitioning is that data is split into divisions that can be accessed faster. The partition key is a column with the values used to split the dataset. Partitioning is useful when there are divisions in your data that are meaningful to work on separately. For example, if your data is based on time intervals, a partition column could be the year value. That way, when a query uses a filter value based on the year, only the data in the partition that matches the requested year is read, instead of the entire dataset.

Partitions can also be nested and are represented by a directory structure in Parquet. So, let's say that we also want to partition by the column month. The folder structure of the Parquet dataset would be similar to the following:

```
hdfs -fs ls /data/data_file

year=2015

year=2016

year=2017

hdfs -fs ls /data/data_file/year=2017

month=01

month=02

month=03

month=04

month=05
```

Partitioning allows better performance when partitions are filtered, as only the data in the chosen partition will be read, increasing performance. To save partitioned files, the **`partitionBy`** option should be used, either in the **`parquet`** command or as the previous command chained to the write operation:

```
df.write.parquet("hdfs:/data/data_file_partitioned", partitionBy=["year",
"month"])
```

The alternative method is:

```
df.write.partitttionBy(["year", "month"]).format("parquet").save("hdfs:/data/
data_file_partitioned")
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The latter format can be used with the previous operations. When reading partitioned data, Spark can infer the partition structure from the directory structure.

An analyst can considerably improve the performance of their queries if partitioning is used correctly. But partitioning can hinder performance if partition columns are not chosen correctly. For example, if there is only one year in the dataset, partitioning per year will not provide any benefits. If there is a column with too many distinct values, partitioning using this column could also create problems, creating too many partitions that would not improve speed and may even slow things down.

Exercise 22: Creating a Partitioned Dataset

We discovered in our preliminary analysis that the data has date columns, one for the year, one for the month, and one for the day. We will be aggregating this data to get the minimum, mean, and maximum values per year, per month, and per day. Let's create a partitioned dataset saved in Parquet from our database:

1. Define a PostgreSQL connection:

```
url = “jdbc:postgresql://posgreserver:5432/timestamped_db”
properties = {“user”: “my_postgre_user”,  password: “mypassword”,
“driver”: “org.postgresql.Driver”}
```

2. Read the data from PostgreSQL to a DataFrame, using the JDBC connector:

```
df = spark.read.jdbc(url, table = “sales”, properties = properties)
```

3. And let's convert this into partitioned Parquet:

```
df.write.parquet(“hdfs:/data/data_file_partitioned”, partitionBy=[“year”,
“month”, “day”], compression=”snappy”)
```

The use of Spark as an intermediary for different data sources, and considering its data processing and transformation capabilities, makes it an excellent tool for combining and analyzing data.

Handling Unstructured Data

Unstructured data usually refers to data that doesn't have a fixed format. CSV files are structured, for example, and JSON files can also be considered structured, although not tabular. Computer logs, on the other hand, don't have the same structure, as different programs and daemons will output messages without a common pattern. Images are also another example of unstructured data, like free text.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

We can leverage Spark's flexibility for reading data to parse unstructured formats and extract the required information into a more structured format, allowing analysis. This step is usually called **pre-processing** or **data wrangling**.

Exercise 23: Parsing Text and Cleaning

In this exercise, we will read a text file, split it into lines and remove the words **the** and **a** from the string given string:

1. Read the text file **shake.txt** (https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson03/data/shake.txt) into the Spark object using the **text** method:

```
from operator import add
rdd_df = spark.read.text("/shake.txt").rdd
```

2. Extract the lines from the text using the following command:

```
lines = rdd_df.map(lambda line: line[0])
```

3. This splits each line in the file as an entry in the list. To check the result, you can use the **collect** method, which gathers all data back to the driver process:

```
lines.collect()
```

4. Now, let's count the number of lines, using the **count** method:

```
lines.count()
```

> **Note**
>
> Be careful when using the **collect** method! If the DataFrame or RDD being collected is larger than the memory of the local driver, Spark will throw an error.

5. Now, let's first split each line into words, breaking it by the space around it, and combining all elements, removing words in uppercase:

```
splits = lines.flatMap(lambda x: x.split(' '))
lower_splits = splits.map(lambda x: x.lower().strip())
```

6. Let's also remove the words **the** and **a**, and punctuations like '.', ',' from the given string:

```
prep = ['the', 'a', ',', '.']
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Use the following command to remove the stop words from our token list:

```
tokens = lower_splits.filter(lambda x: x and x not in prep)
```

   We can now process our token list and count the unique words. The idea is to generate a list of tuples, where the first element is the token and the second element is the count of that particular token.

8. Let's map our token to a list:

```
token_list = tokens.map(lambda x: [x, 1])
```

9. Use the **reduceByKey** operation, which will apply the operation to each of the lists:

```
count = token_list.reduceByKey(add).sortBy(lambda x: x[1],
ascending=False)
count.collect()
```

   The output is as follows:

```
>>> from operator import add
>>> rdd_df = spark.read.text("shake.txt").rdd
>>> lines = rdd_df.map(lambda line: line[0])
>>> lines.collect()
['It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it wa
s the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it
 was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were
 all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present
period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlativ
e degree of comparison only.']
>>>
>>> splits = lines.flatMap(lambda x: x.split(' '))
>>> lower_splits = splits.map(lambda x: x.lower().strip())
>>> prep = ['the', 'a', ',', '.']
>>> tokens = lower_splits.filter(lambda x: x and x not in prep)
>>> token_list = tokens.map(lambda x: [x, 1])
>>> count = token_list.reduceByKey(add).sortBy(lambda x: x[1], ascending=False)
>>> count.collect()
[('of', 12), ('was', 11), ('it', 10), ('we', 4), ('times,', 2), ('age', 2), ('epoch', 2), ('season', 2), ('had', 2),
('before', 2), ('us,', 2), ('were', 2), ('all', 2), ('going', 2), ('direct', 2), ('its', 2), ('for', 2), ('best', 1),
 ('worst', 1), ('wisdom,', 1), ('foolishness,', 1), ('belief,', 1), ('incredulity,', 1), ('light,', 1), ('darkness,',
 1), ('spring', 1), ('hope,', 1), ('winter', 1), ('despair,', 1), ('everything', 1), ('nothing', 1), ('to', 1), ('hea
ven,', 1), ('other', 1), ('way—in', 1), ('short,', 1), ('period', 1), ('so', 1), ('far', 1), ('like', 1), ('present',
 1), ('period,', 1), ('that', 1), ('some', 1), ('noisiest', 1), ('authorities', 1), ('insisted', 1), ('on', 1), ('bei
ng', 1), ('received,', 1), ('good', 1), ('or', 1), ('evil,', 1), ('in', 1), ('superlative', 1), ('degree', 1), ('comp
arison', 1), ('only.', 1)]
>>>
```

Figure 3.3: Parsing Text and Cleaning

> **Note**
>
> Remember, **collect()**collects all data back to the driver node! Always check whether there is enough memory by using tools such as **top** and **htop**.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 8: Removing Stop Words from Text

In this activity, we will read a text file, split it into lines and remove the **`stopwords`** from the text:

1. Read the text file **`shake.txt`** as used in Exercise 8.
2. Extract the lines from the text and create a list with each line.
3. Split each line into words, breaking it by the space around it and remove words in uppercase.
4. Remove the stop words: ‘of’, ‘a’, ‘and’, ‘to’ from our token list.
5. Process the token list and count the unique words, generating list of tuples made up of the token and its count.
6. Map our tokens to a list using the **`reduceByKey`** operation.

   The output is as follows:

```
bin : python — Konsole
('before', 2), ('us,', 2), ('were', 2), ('all', 2), ('going', 2), ('direct', 2), ('its', 2), ('for', 2), ('best', 1),
 ('worst', 1), ('wisdom,', 1), ('foolishness,', 1), ('belief,', 1), ('incredulity,', 1), ('light,', 1), ('darkness,',
 1), ('spring', 1), ('hope,', 1), ('winter', 1), ('despair,', 1), ('everything', 1), ('nothing', 1), ('to', 1), ('hea
ven,', 1), ('other', 1), ('way—in', 1), ('short,', 1), ('period', 1), ('so', 1), ('far', 1), ('like', 1), ('present',
 1), ('period,', 1), ('that', 1), ('some', 1), ('noisiest', 1), ('authorities', 1), ('insisted', 1), ('on', 1), ('bei
ng', 1), ('received,', 1), ('good', 1), ('or', 1), ('evil,', 1), ('in', 1), ('superlative', 1), ('degree', 1), ('comp
arison', 1), ('only.', 1)]
>>> from operator import add
>>> rdd_df = spark.read.text("shake.txt").rdd
>>> lines = rdd_df.map(lambda line: line[0])
>>> lines.collect()
['It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it wa
s the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it
 was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were
 all going direct to Heaven, we were all going direct the other way—in short, the period was so far like the present
period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlativ
e degree of comparison only.']
>>>
>>> splits = lines.flatMap(lambda x: x.split(' '))
>>> lower_splits = splits.map(lambda x: x.lower().strip())
>>> #prep = ['the', 'a', ',', '.', 'of', 'a', 'and', 'to']
... prep = ['of', 'a', 'and', 'to']
>>> tokens = lower_splits.filter(lambda x: x and x not in prep)
>>> token_list = tokens.map(lambda x: [x, 1])
>>> count = token_list.reduceByKey(add).sortBy(lambda x: x[1], ascending=False)
>>> count.collect()
[('the', 14), ('was', 11), ('it', 10), ('we', 4), ('times,', 2), ('age', 2), ('epoch', 2), ('season', 2), ('had', 2),
 ('before', 2), ('us,', 2), ('were', 2), ('all', 2), ('going', 2), ('direct', 2), ('its', 2), ('for', 2), ('best', 1)
, ('worst', 1), ('wisdom,', 1), ('foolishness,', 1), ('belief,', 1), ('incredulity,', 1), ('light,', 1), ('darkness,'
, 1), ('spring', 1), ('hope,', 1), ('winter', 1), ('despair,', 1), ('everything', 1), ('nothing', 1), ('heaven,', 1),
 ('other', 1), ('way—in', 1), ('short,', 1), ('period', 1), ('so', 1), ('far', 1), ('like', 1), ('present', 1), ('per
iod,', 1), ('that', 1), ('some', 1), ('noisiest', 1), ('authorities', 1), ('insisted', 1), ('on', 1), ('being', 1), (
'received,', 1), ('good', 1), ('or', 1), ('evil,', 1), ('in', 1), ('superlative', 1), ('degree', 1), ('comparison', 1
), ('only.', 1)]
>>>
```

Figure 3.4: Removing Stop Words from Text

> **Note**
>
> The solution for this activity can be found on page 213.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

We get the list of tuples, where each tuple is a token and the count of the number of times that word appeared in the text. Notice that, before the final collect on count (an action), the operations that were transformations did not start running right away: we needed the action operation count to Spark start executing all the steps.

Other kinds of unstructured data can be parsed using the preceding example, and either operated on directly, such as in the preceding activity or transformed into a DataFrame later.

Summary

After a review of what big data is, we learned about some tools that were designed for the storage and processing of very large volumes of data. Hadoop is an entire ecosystem of frameworks and tools, such as HDFS, designed to store data in a distributed fashion in a huge number of commodity-computing nodes, and YARN, a resource and job manager. We saw how to manipulate data directly on the HDFS using the HDFS fs commands.

We also learned about Spark, a very powerful and flexible parallel processing framework that integrates well with Hadoop. Spark has different APIs, such as SQL, GraphX, and Streaming. We learned how Spark represents data in the DataFrame API and that its computation is similar to pandas' methods. We also saw how to store data in an efficient manner using the Parquet file format, and how to improve performance when analyzing data using partitioning. To finish up, we saw how to handle unstructured data files, such as text.

In the next chapter, we will go more deeply into how to create a meaningful statistical analysis using more advanced techniques with Spark and how to use Jupyter notebooks with Spark.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

4

Diving Deeper with Spark

Learning Objectives

By the end of this chapter, you will be able to:

- Implement the basic Spark DataFrame API
- Read data and create Spark DataFrames from different data sources
- Manipulate and process data using different Spark DataFrame options
- Visualize data in Spark DataFrames using different plots

In this chapter, we will use the Spark as an analysis tool for big datasets.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

The last chapter introduced us to one of the most popular distributed data processing platforms used to process big data–Spark.

In this chapter, we will learn more about how to work with Spark and Spark DataFrames using its Python API–**PySpark**. It gives us the capability to process petabyte-scale data, but also implements **machine learning** (**ML**) algorithms at petabyte scale in real time. This chapter will focus on the data processing part using Spark DataFrames in PySpark.

> **Note**
>
> We will be using the term DataFrame quite frequently during this chapter. This will explicitly refer to the Spark DataFrame, unless mentioned otherwise. Please do not confuse this with the pandas DataFrame.

Spark DataFrames are a distributed collection of data organized as named columns. They are inspired from R and Python DataFrames and have complex optimizations at the backend that make them fast, optimized, and scalable.

The DataFrame API was developed as part of **Project Tungsten** and is designed to improve the performance and scalability of Spark. It was first introduced with Spark 1.3.

Spark DataFrames are much easier to use and manipulate than their predecessor RDDs. They are *immutable*, like RDDs, and support lazy loading, which means no transformation is performed on the DataFrames unless an action is called. The execution plan for the DataFrames is prepared by Spark itself and hence is more optimized, making operations on DataFrames faster than those on RDDs.

Getting Started with Spark DataFrames

To get started with Spark DataFrames, we will have to create something called a SparkContext first. SparkContext configures the internal services under the hood and facilitates command execution from the Spark execution environment.

> **Note**
>
> We will be using Spark version 2.1.1, running on Python 3.7.1. Spark and Python are installed on a MacBook Pro, running macOS Mojave version 10.14.3, with a 2.7 GHz Intel Core i5 processor and 8 GB 1867 MHz DDR3 RAM.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The following code snippet is used to create **SparkContext**:

```
from pyspark import SparkContext
sc = SparkContext()
```

> **Note**
>
> In case you are working in the PySpark shell, you should skip this step, as the shell automatically creates the **sc (SparkContext)** variable when it is started. However, be sure to create the **sc** variable while creating a PySpark script or working with Jupyter Notebook, or your code will throw an error.

We also need to create an **SQLContext** before we can start working with DataFrames. **SQLContext** in Spark is a class that provides SQL-like functionality within Spark. We can create **SQLContext** using **SparkContext**:

```
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

There are three different ways of creating a DataFrame in Spark:

- We can programmatically specify the schema of the DataFrame and manually enter the data in it. However, since Spark is generally used to handle big data, this method is of little use, apart from creating data for small test/sample cases.
- Another method to create a DataFrame is from an existing RDD object in Spark. This is useful, because working on a DataFrame is way easier than working directly with RDDs.
- We can also read the data directly from a data source to create a Spark DataFrame. Spark supports a variety of external data sources, including CSV, JSON, parquet,

  RDBMS tables, and Hive tables.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 24: Specifying the Schema of a DataFrame

In this exercise, we will create a small sample DataFrame by manually specifying the schema and entering data in Spark. Even though this method has little application in a practical scenario, it will be a good starting point in getting started with Spark DataFrames:

1. Importing the necessary files:

   ```
   from pyspark import SparkContext
   sc = SparkContext()
   from pyspark.sql import SQLContext
   sqlc = SQLContext(sc)
   ```

2. Import SQL utilities from the PySpark module and specify the schema of the sample DataFrame:

   ```
   from pyspark.sql import *
   na_schema = Row("Name","Age")
   ```

3. Create rows for the DataFrame as per the specified schema:

   ```
   row1 = na_schema("Ankit", 23)
   row2 = na_schema("Tyler", 26)
   row3 = na_schema("Preity", 36)
   ```

4. Combine the rows together to create the DataFrame:

   ```
   na_list = [row1, row2, row3]
   df_na = sqlc.createDataFrame(na_list)
   type(df_na)
   ```

5. Now, show the DataFrame using the following command:

   ```
   df_na.show()
   ```

   The output is as follows:

   ```
   +------+---+
   |  Name|Age|
   +------+---+
   | Ankit| 23|
   | Tyler| 26|
   |Preity| 36|
   +------+---+
   ```

Figure 4.1: Sample PySpark DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 25: Creating a DataFrame from an Existing RDD

In this exercise, we will create a small sample DataFrame from an existing RDD object in Spark:

1. Create an RDD object that we will convert into DataFrame:

```
data = [("Ankit",23),("Tyler",26),("Preity",36)]
data_rdd = sc.parallelize(data)
type(data_rdd)
```

2. Convert the RDD object into a DataFrame:

```
data_sd = sqlc.createDataFrame(data_rdd)
```

3. Now, show the DataFrame using the following command:

```
data_sd.show()
```

```
+------+---+
|    _1| _2|
+------+---+
| Ankit| 23|
| Tyler| 26|
|Preity| 36|
+------+---+
```

Figure 4.2: DataFrame converted from the RDD object

Exercise 25: Creating a DataFrame Using a CSV File

A variety of different data sources can be used to create a DataFrame. In this exercise, we will use the open source Iris dataset, which can be found under datasets in the scikit-learn library. The Iris dataset is a multivariate dataset containing 150 records, with 50 records for each of the 3 species of Iris flower (Iris Setosa, Iris Virginica, and Iris Versicolor).

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The dataset contains five attributes for each of the Iris species, namely, **`petal length`**, **`petal width`**, **`sepal length`**, **`sepal width`**, and **`species`**. We have stored this dataset in an external CSV file that we will read into Spark:

1. Download and install the PySpark CSV reader package from the Databricks website:

```
pyspark -packages com.databricks:spark-csv_2.10:1.4.0
```

2. Read the data from the CSV file into the Spark DataFrame:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header='true',
inferschema='true').load('iris.csv')
type(df)
```

3. Now, show the DataFrame using the following command:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.3: Iris DataFrame, first four rows

> **Note for the Instructor**
>
> Motivate the students to explore other data sources, such as tab-separated files, parquet files, and relational databases, as well.

Writing Output from Spark DataFrames

Spark gives us the ability to write the data stored in Spark DataFrames into a local pandas DataFrame, or write them into external structured file formats such as CSV. However, before converting a Spark DataFrame into a local pandas DataFrame, make sure that the data would fit in the local driver memory.

In the following exercise, we will explore how to convert the Spark DataFrame to a pandas DataFrame.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 27: Converting a Spark DataFrame to a Pandas DataFrame

In this exercise, we will use the pre-created Spark DataFrame of the Iris dataset in the previous exercise, and convert it into a local pandas DataFrame. We will then store this DataFrame into a CSV file. Perform the following steps:

1. Convert the Spark DataFrame into a pandas DataFrame using the following command:

```
import pandas as pd
df.toPandas()
```

2. Now use the following command to write the pandas DataFrame to a CSV file:

```
df.toPandas().to_csv('iris.csv')
```

> **Note**
>
> Writing the contents of a Spark DataFrame to a CSV file requires a one-liner using the **spark-csv** package:
>
> ```
> df.write.csv('iris.csv')
> ```

Exploring Spark DataFrames

One of the major advantages that the Spark DataFrames offer over the traditional RDDs is the ease of data use and exploration. The data is stored in a more structured tabular format in the DataFrames and hence is easier to make sense of. We can compute basic statistics such as the number of rows and columns, look at the schema, and compute summary statistics such as mean and standard deviation.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 28: Displaying Basic DataFrame Statistics

In this exercise, we will show basic DataFrame statistics of the first few rows of the data, and summary statistics for all the numerical DataFrame columns and an individual DataFrame column:

1. Look at the DataFrame schema. The schema is displayed in a tree format on the console:

```
df.printSchema()
```

```
root
 |-- Sepallength: double (nullable = true)
 |-- Sepalwidth: double (nullable = true)
 |-- Petallength: double (nullable = true)
 |-- Petalwidth: double (nullable = true)
 |-- Species: string (nullable = true)
```

Figure 4.4: Iris DataFrame schema

2. Now, use the following command to print the column names of the Spark DataFrame:

```
df.schema.names
```

```
['Sepallength', 'Sepalwidth', 'Petallength', 'Petalwidth', 'Species']
```

Figure 4.5: Iris column names

3. To retrieve the number of rows and columns present in the Spark DataFrame, use the following command:

```
## Counting the number of rows in DataFrame
df.count()#134

## Counting the number of columns in DataFrame
len(df.columns)#5
```

4. Let's fetch the first *n* rows of the data. We can do this by using the **`head()`** method. However, we use the **`show()`** method as it displays the data in a better format:

```
df.show(4)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.6: Iris DataFrame, first four rows

5. Now, compute the summary statistics, such as mean and standard deviation, for all the numerical columns in the DataFrame:

```
df.describe().show()
```

The output is as follows:

```
+-------+------------------+-------------------+------------------+------------------+---------+
|summary|       Sepallength|         Sepalwidth|       Petallength|        Petalwidth|  Species|
+-------+------------------+-------------------+------------------+------------------+---------+
|  count|               148|                150|               149|               150|      150|
|   mean| 5.854729729729732|  3.057333333333334|3.7744966442953043| 1.199333333333334|     null|
| stddev|0.8277774898579762|0.43586628493669793|1.7596127630823133|0.7622376689603467|     null|
|    min|               4.3|                2.0|               1.0|               0.1|   setosa|
|    max|               7.9|                4.4|               6.9|               2.5|virginica|
+-------+------------------+-------------------+------------------+------------------+---------+
```

Figure 4.7: Iris DataFrame, summary statistics

6. To compute the summary statistics for an individual numerical column of a Spark DataFrame, use the following command:

```
df.describe('Sepalwidth').show()
```

The output is as follows:

```
+-------+-------------------+
|summary|         Sepalwidth|
+-------+-------------------+
|  count|                150|
|   mean|  3.057333333333334|
| stddev|0.43586628493669793|
|    min|                2.0|
|    max|                4.4|
+-------+-------------------+
```

Figure 4.8: Iris DataFrame, summary statistics of Sepalwidth column

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 9: Getting Started with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections and create a Spark DataFrame using all three methods. We will also compute DataFrame statistics, and finally, write the same data into a CSV file. Feel free to use any open source dataset for this activity:

1. Create a sample DataFrame by manually specifying the schema.
2. Create a sample DataFrame from an existing RDD.
3. Create a sample DataFrame by reading the data from a CSV file.
4. Print the first seven rows of the sample DataFrame read in step 3.
5. Print the schema of the sample DataFrame read in step 3.
6. Print the number of rows and columns in the sample DataFrame.
7. Print the summary statistics of the DataFrame and any 2 individual numerical columns.
8. Write the first 7 rows of the sample DataFrame to a CSV file using both methods mentioned in the exercises.

> **Note**
>
> The solution for this activity can be found on page 215.

Data Manipulation with Spark DataFrames

Data manipulation is a prerequisite for any data analysis. To draw meaningful insights from the data, we first need to understand, process, and massage the data. But this step becomes particularly hard with the increase in the size of data. Due to the scale of data, even simple operations such as filtering and sorting become complex coding problems. Spark DataFrames make data manipulation on big data a piece of cake.

Manipulating the data in Spark DataFrames is quite like working on regular pandas DataFrames. Most of the data manipulation operations on Spark DataFrames can be done using simple and intuitive one-liners. We will use the Spark DataFrame containing the Iris dataset that we created in previous exercises for these data manipulation exercises.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 29: Selecting and Renaming Columns from the DataFrame

In this exercise, we will first rename the column using the **withColumnRenamed** method and then select and print the schema using the **select** method.

Perform the following steps:

1. Rename the columns of a Spark DataFrame using the **withColumnRenamed()** method:

```
df = df.withColumnRenamed('Sepal.Width','Sepalwidth')
```

> **Note**
>
> Spark does not recognize column names containing a period(.). Make sure to rename them using this method.

2. Select a single column or multiple columns from a Spark DataFrame using the **select** method:

```
df.select('Sepalwidth','Sepallength').show(4)
```

```
+----------+-----------+
|Sepalwidth|Sepallength|
+----------+-----------+
|       3.5|        5.1|
|       3.0|        4.9|
|       3.2|        4.7|
|       3.1|        4.6|
+----------+-----------+
only showing top 4 rows
```

Figure 4.9: Iris DataFrame, Sepalwidth and Sepallength column

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 30: Adding and Removing a Column from the DataFrame

In this exercise, we will add a new column in the dataset using the **withColumn** method, and later, using the **drop** function, will remove it. Now, let's perform the following steps:

1. Add a new column in a Spark DataFrame using the **withColumn** method:

```
df = df.withColumn('Half_sepal_width', df['Sepalwidth']/2.0)
```

2. Use the following command to show the dataset with the newly added column:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+----------------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|Half_sepal_width|
+-----------+----------+-----------+----------+-------+----------------+
|        5.1|       3.5|        1.4|       0.2| setosa|            1.75|
|        4.9|       3.0|        1.4|       0.2| setosa|             1.5|
|        4.7|       3.2|        1.3|       0.2| setosa|             1.6|
|        4.6|       3.1|        1.5|       0.2| setosa|            1.55|
+-----------+----------+-----------+----------+-------+----------------+
only showing top 4 rows
```

Figure 4.10: Introducing new column, Half_sepal_width

3. Now, to remove a column in a Spark DataFrame, use the **drop** method illustrated here:

```
df = df.drop('Half_sepal_width')
```

4. Let's show the dataset to verify that the column has been removed:

```
df.show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.11: Iris DataFrame after dropping the Half_sepal_width column

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 31: Displaying and Counting Distinct Values in a DataFrame

To display the distinct values in a DataFrame, we use the **distinct().show()** method. Similarly, to count the distinct values, we will be using the **distinct().count()** method. Perform the following procedures to print the distinct values with the total count:

1. Select the distinct values in any column of a Spark DataFrame using the **distinct** method, in conjunction with the **select** method:

```
df.select('Species').distinct().show()
```

```
+----------+
|   Species|
+----------+
| virginica|
|versicolor|
|    setosa|
+----------+
```

Figure 4.12: Iris DataFrame, Species column

2. To count the distinct values in any column of a Spark DataFrame, use the **count** method, in conjunction with the **distinct** method:

```
df.select('Species').distinct().count()
```

Exercise 32: Removing Duplicate Rows and Filtering Rows of a DataFrame

In this exercise, we will learn how to remove the duplicate rows from the dataset, and later, perform filtering operations on the same column.

Perform these steps:

1. Remove the duplicate values from a DataFrame using the **dropDuplicates()** method:

```
df.select('Species').dropDuplicates().show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
+----------+
|   Species|
+----------+
| virginica|
|versicolor|
|    setosa|
+----------+
```

Figure 4.13: Iris DataFrame, Species column after removing duplicate column

2. Filter the rows from a DataFrame using one or multiple conditions. These multiple conditions can be passed together to the DataFrame using Boolean operators such as and (**&**), or **|**, similar to how we do it for pandas DataFrames:

```
# Filtering using a single condition
df.filter(df.Species == 'setosa').show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 4 rows
```

Figure 4.14: Iris DataFrame after filtering with single conditions

3. Now, to filter the column using multiple conditions, use the following command:

```
df.filter((df.Sepallength > 5) & (df.Species == 'setosa')).show(4)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|       null|       3.4|        1.6|       0.4| setosa|
|       null|       3.0|        1.6|       0.2| setosa|
|        4.3|       3.0|        1.1|       0.1| setosa|
|        4.4|       3.0|       null|       0.2| setosa|
|        4.4|       2.9|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 4.15: Iris DataFrame after filtering with multiple conditions

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 33: Ordering Rows in a DataFrame

In this exercise, we will explore how to sort the rows in a DataFrame in ascending and descending order. Let's perform these steps:

1. Sort the rows in a DataFrame, using one or multiple conditions, in ascending or descending order:

```
df.orderBy(df.Sepallength).show(5)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|       null|       3.4|        1.6|       0.4| setosa|
|       null|       3.0|        1.6|       0.2| setosa|
|        4.3|       3.0|        1.1|       0.1| setosa|
|        4.4|       3.0|       null|       0.2| setosa|
|        4.4|       2.9|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 4.16: Filtered Iris DataFrame

2. To sort the rows in descending order, use the following command:

```
df.orderBy(df.Sepallength.desc()).show(5)
```

```
+-----------+----------+-----------+----------+---------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|  Species|
+-----------+----------+-----------+----------+---------+
|        7.9|       3.8|        6.4|       2.0|virginica|
|        7.7|       2.6|        6.9|       2.3|virginica|
|        7.7|       3.8|        6.7|       2.2|virginica|
|        7.7|       3.0|        6.1|       2.3|virginica|
|        7.7|       2.8|        6.7|       2.0|virginica|
+-----------+----------+-----------+----------+---------+
only showing top 5 rows
```

Figure 4.17: Iris DataFrame after sorting it in the descending order

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 34: Aggregating Values in a DataFrame

We can group the values in a DataFrame by one or more variable, and calculate aggregated metrics such as **mean**, **sum**, **count**, and many more. In this exercise, we will calculate the mean sepal width for each of the flower species in the Iris dataset. We will also calculate the count of the rows for each species:

1. To calculate the mean sepal width for each species, use the following command:

```
df.groupby('Species').agg({'Sepalwidth' : 'mean'}).show()
```

```
+----------+------------------+
|   Species|   avg(Sepalwidth)|
+----------+------------------+
| virginica|2.9739999999999998|
|versicolor|2.7700000000000005|
|    setosa| 3.428000000000001|
+----------+------------------+
```

Figure 4.18: Iris DataFrame, calculating mean sepal width

2. Now, let's calculate the number of rows for each species by using the following command:

```
df.groupby('Species').count().show()
```

```
+----------+-----+
|   Species|count|
+----------+-----+
| virginica|   50|
|versicolor|   50|
|    setosa|   50|
+----------+-----+
```

Figure 4.19: Iris DataFrame, calculating number of rows for each species

> **Note**
>
> In the second code snippet, the count can also be used with the **.agg** function; however, the method we used is more popular.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 10: Data Manipulation with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections to manipulate the data in the Spark DataFrame created using the Iris dataset. We will perform basic data manipulation steps to test our ability to work with data in a Spark DataFrame. Feel free to use any open source dataset for this activity. Make sure the dataset you use has both numerical and categorical variables:

1. Rename any five columns of the DataFrame. If the DataFrame has more than columns, rename all the columns.
2. Select two numeric and one categorical column from the DataFrame.
3. Count the number of distinct categories in the categorical variable.
4. Create two new columns in the DataFrame by summing up and multiplying together the two numerical columns.
5. Drop both the original numerical columns.
6. Sort the data by the categorical column.
7. Calculate the mean of the summation column for each distinct category in the categorical variable.
8. Filter the rows with values greater than the mean of all the mean values calculated in step 7.
9. De-duplicate the resultant DataFrame to make sure it has only unique records.

> **Note**
>
> The solution for this activity can be found on page 219.

Graphs in Spark

The ability to effectively visualize data is of paramount importance. Visual representations of data help the user develop a better understanding of data and uncover trends that might go unnoticed in text form. There are numerous types of plots available in Python, each with its own context.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will be exploring some of these plots, including bar charts, density plots, boxplots, and linear plots for Spark DataFrames, using the widely used Python plotting packages of Matplotlib and Seaborn. The point to note here is that Spark deals with big data. So, make sure that your data size is reasonable enough (that is, it fits in your computer's RAM) before plotting it. This can be achieved by filtering, aggregating, or sampling the data before plotting it.

We are using the Iris dataset, which is small, hence we do not need to do any such pre-processing steps to reduce the data size.

> **Note for the Instructor**
>
> The user should install and load the Matplotlib and Seaborn packages beforehand, in the development environment, before getting started with the exercises in this section. If you are unfamiliar with installing and loading these packages, visit the official websites of Matplotlib and Seaborn.

Exercise 35: Creating a Bar Chart

In this exercise, we will try to plot the number of records available for each species using a bar chart. We will have to first aggregate the data and count the number of records for each species. We can then convert this aggregated data into a regular pandas DataFrame and use Matplotlib and Seaborn packages to create any kind of plots of it that we wish:

1. First, calculate the number of rows for each flower species and convert the result to a pandas DataFrame:

```
data = df.groupby('Species').count().toPandas()
```

2. Now, create a bar plot from the resulting pandas DataFrame:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.barplot( x = data['Species'], y = data['count'])
plt.xlabel('Species')
plt.ylabel('count')
plt.title('Number of rows per species')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The plot is as follows:

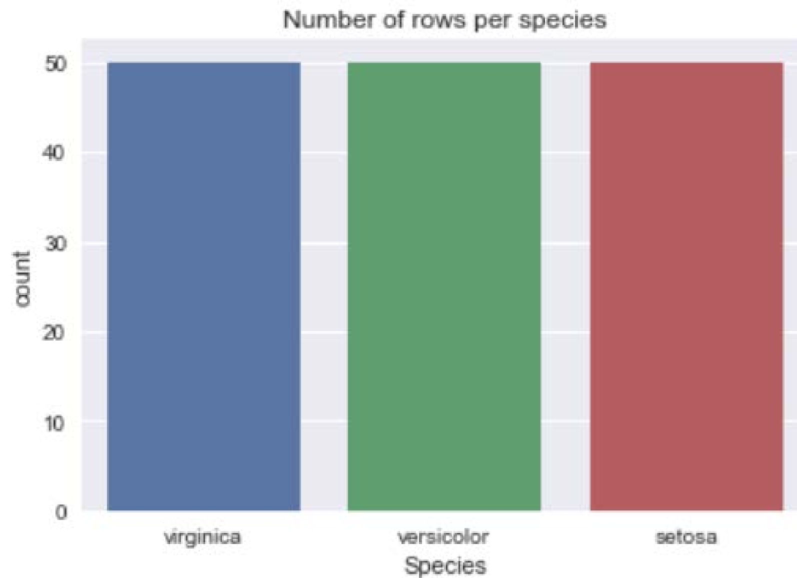


Figure 4.20: Bar plot for Iris DataFrame after calculating the number of rows for each flower species

Exercise 36: Creating a Linear Model Plot

In this exercise, we will plot the data points of two different variables and fit a straight line on them. This is similar to fitting a linear model on two variables and can help identify correlations between the two variables:

1. Create a **data** object from the pandas DataFrame:

```
data = df.toPandas()
sns.lmplot(x = "Sepallength", y = "Sepalwidth", data = data)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Plot the DataFrame using the following command:

```
plt.show()
```

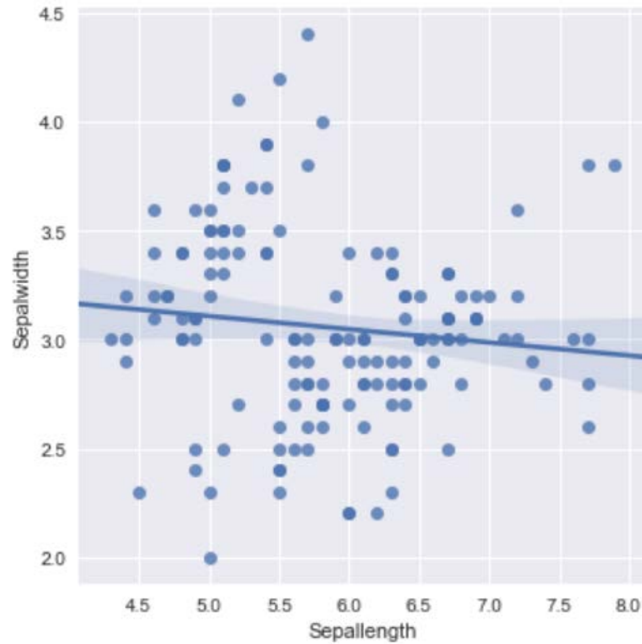


Figure 4.21: Linear model plot for Iris DataFrame

Exercise 37: Creating a KDE Plot and a Boxplot

In this exercise, we will create a **kernel density estimation** (**KDE**) plot, followed by a **boxplot**. Follow these instructions:

1. First, plot a KDE plot that shows us the distribution of a variable. Make sure it gives us an idea of the skewness and the kurtosis of a variable:

```
import seaborn as sns
data = df.toPandas()
sns.kdeplot(data.Sepalwidth, shade = True)
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

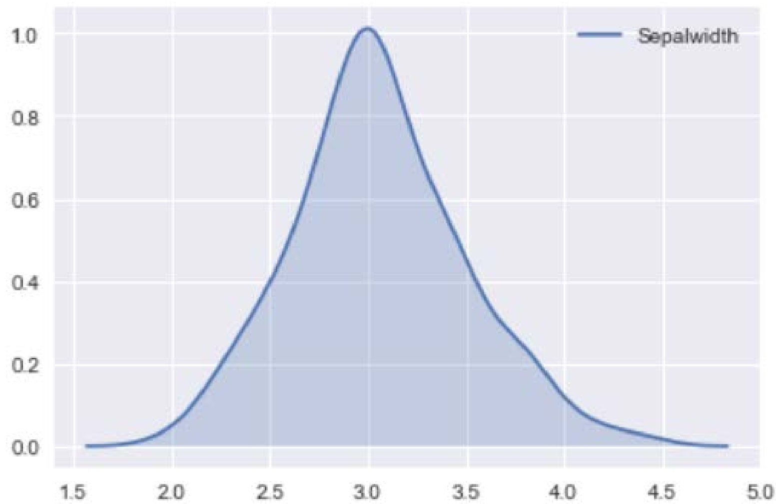


Figure 4.22: KDE plot for the Iris DataFrame

2. Now, plot the boxplots for the Iris dataset using the following command:

```
sns.boxplot(x = "Sepallength", y = "Sepalwidth", data = data)
plt.show()
```

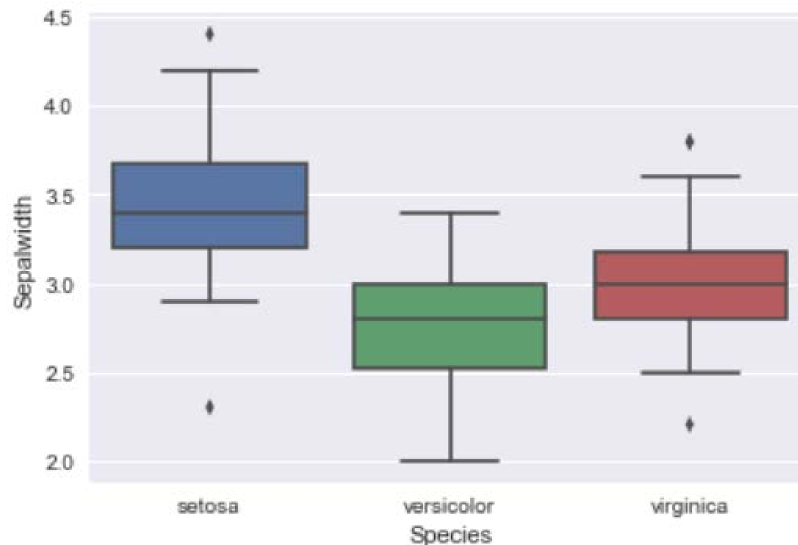


Figure 4.23: Boxplot for the Iris DataFrame

Boxplots are a good way to look at the data distribution and locate outliers. They represent the distribution using the 1st quartile, the median, the 3rd quartile, and the interquartile range (25th to 75th percentile).

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 11: Graphs in Spark

In this activity, we will use the plotting libraries of Python to visually explore our data using different kind of plots. For this activity, we are using the **mtcars** dataset from Kaggle (https://www.kaggle.com/ruiromanini/mtcars):

1. Import all the required packages and libraries in the Jupyter Notebook.
2. Read the data into Spark object from the **mtcars** dataset.
3. Visualize the discrete frequency distribution of any continuous numeric variable from your dataset using a histogram:

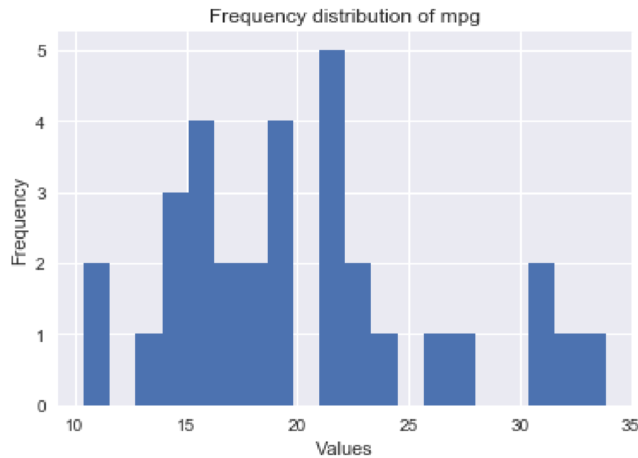


Figure 4.24: Histogram for the Iris DataFrame

4. Visualize the percentage share of the categories in the dataset using a pie chart:

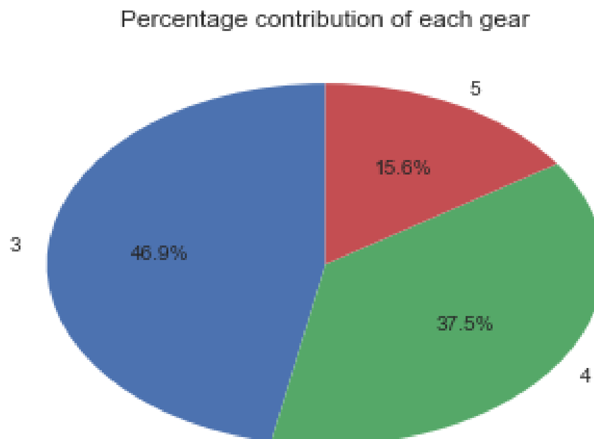


Figure 4.25: Pie chart for the Iris DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Plot the distribution of a continuous variable across the categories of a categorical variable using a boxplot:

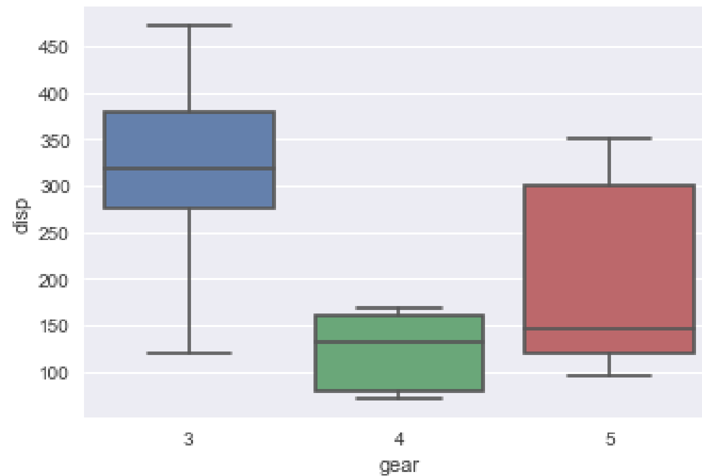


Figure 4.26: Boxplot for the Iris DataFrame

6. Visualize the values of a continuous numeric variable using a line chart:

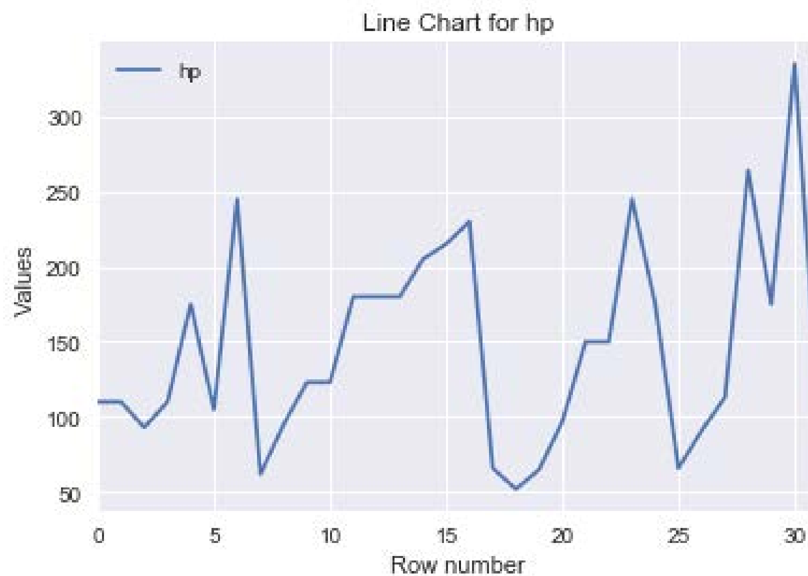


Figure 4.27: Line chart for the Iris DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Plot the values of multiple continuous numeric variables on the same line chart:

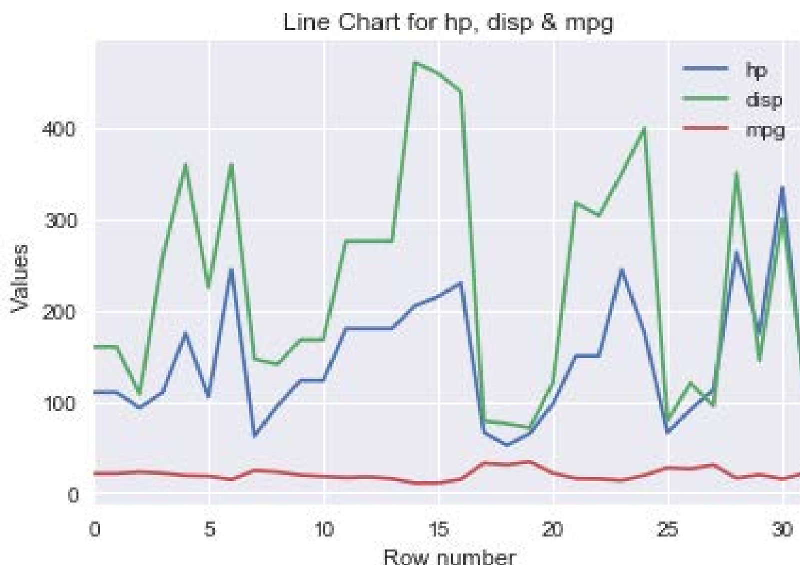


Figure 4.28: Line chart for the Iris DataFrame plotting multiple continuous numeric variables

> **Note**
>
> The solution for this activity can be found on page 224.

Summary

In this chapter, we saw a basic introduction of Spark DataFrames and how they are better than RDDs. We explored different ways of creating Spark DataFrames and writing the contents of Spark DataFrames to regular pandas DataFrames and output files.

We tried out hands-on data exploration in PySpark by computing basic statistics and metrics for Spark DataFrames. We played around with the data in Spark DataFrames and performed data manipulation operations such as filtering, selection, and aggregation. We tried our hands at plotting the data to generate insightful visualizations.

Furthermore, we consolidated our understanding of various concepts by practicing hands-on exercises and activities.

In the next chapter, we will explore how to handle missing values and compute correlation between variables in PySpark.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5

Handling Missing Values and Correlation Analysis

Learning Objectives

By the end of this chapter, you will be able to:

- Detect and handle missing values in data using PySpark
- Describe correlations between variables
- Compute correlations between two or more variables in PySpark
- Create a correlation matrix using PySpark

In this chapter, we will be using the Iris dataset to handle missing data and find correlations between data values.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

In the previous chapter, we learned the basic concepts of Spark DataFrames and saw how to leverage them for big data analysis.

In this chapter, we will go a step further and learn about handling missing values in data and correlation analysis with Spark DataFrames—concepts that will help us with data preparation for machine learning and exploratory data analysis.

We will briefly cover these concepts to provide the reader with some context, but our focus is on their implementation with Spark DataFrames. We will use the same Iris dataset that we used in the previous chapter for the exercises in this chapter as well. But the Iris dataset has no missing values, so we have randomly removed two entries from the **`Sepallength`** column and one entry from the **`Petallength`** column from the original dataset. So, now we have a dataset with missing values, and we will learn how to handle these missing values using PySpark.

We will also look at the correlation between the variables in the Iris dataset by computing their correlation coefficients and correlation matrix.

Setting up the Jupyter Notebook

The following steps are required before getting started with the exercises:

1. Import all the required modules and packages in the Jupyter notebook:

```
import findspark
findspark.init()
import pyspark
import random
```

2. Now, use the following command to set up **`SparkContext`**:

```
from pyspark import SparkContext
sc = SparkContext()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Similarly, use the following command to set up **SQLContext** in the Jupyter notebook:

```
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

> **Note**
>
> Make sure you have the PySpark CSV reader package from the Databricks website (https://databricks.com/) installed and ready before executing the next command. If not, then download it using the following command:
>
> **pyspark -packages com.databricks:spark-csv_2.10:1.4.0**

4. Read the Iris dataset from the CSV file into a Spark DataFrame:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header = 'true',
inferschema = 'true').load('/Users/iris.csv')
```

The output of the preceding command is as follows:

```
df.show(5)
```

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
|        5.0|       3.6|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 5.1: Iris DataFrame

Missing Values

The data entries with no value assigned to them are called **missing values**. In the real world, encountering missing values in data is common. Values may be missing for a wide variety of reasons, such as non-responsiveness of the system/responder, data corruption, and partial deletion.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Some fields are more likely than other fields to contain missing values. For example, income data collected from surveys is likely to contain missing values, because of people not wanting to disclose their income.

Nevertheless, it is one of the major problems plaguing the data analytics world. Depending on the percentage of missing data, missing values may prove to be a significant challenge in data preparation and exploratory analysis. So, it's important to calculate the missing data percentage before getting started with data analysis.

In the following exercise, we will learn how to detect and calculate the number of missing value entries in PySpark DataFrames.

Exercise 38: Counting Missing Values in a DataFrame

In this exercise, we will learn how to count the missing values from the PySpark DataFrame column:

1. Use the following command to check whether the Spark DataFrame has missing values or not:

```
from pyspark.sql.functions import isnan, when, count, col
df.select([count(when(isnan(c) | col(c).isNull(),
                c)).alias(c) for c in df.columns]).show()
```

2. Now, we will count the missing values in the **`Sepallength`** column of the Iris dataset loaded in the PySpark DataFrame **`df`** object:

```
df.filter(col('Sepallength').isNull()).count()
```

The output is as follows:

```
2
```

Exercise 39: Counting Missing Values in All DataFrame Columns

In this exercise, we will count the missing values present in all the columns of a PySpark DataFrame:

1. First, import all the required modules, as illustrated here:

```
from pyspark.sql.functions import isnan, when, count, col
```

2. Now let's show the data using the following command:

```
df.select([count(when(isnan(i) | col(i).isNull(), i)).alias(i) for i in
df.columns]).show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|          2|         0|          1|         0|      0|
+-----------+----------+-----------+----------+-------+
```

Figure 5.2: Iris DataFrame, counting missing values

The output shows we have **2** missing entries in the **Seapllength** column and **1** missing entry in the **Petallength** column in the PySpark DataFrame.

3. A simple way is to just use the **describe()** function, which gives the count of non-missing values for each column, along with a bunch of other summary statistics. Let's execute the following command in the notebook:

```
df.describe().show(1)
```

```
+-------+-----------+----------+-----------+----------+-------+
|summary|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-------+-----------+----------+-----------+----------+-------+
|  count|        148|       150|        149|       150|    150|
+-------+-----------+----------+-----------+----------+-------+
only showing top 1 row
```

Figure 5.3: Iris DataFrame, counting the missing values using different approach

As we can see, there are **148** non-missing values in the **Sepallength** column, indicating **2** missing entries and **149** non-missing values in the **Petallength** column, indicating **1** missing entry.

In the following section, we will explore how to find the missing values from the DataFrame.

Fetching Missing Value Records from the DataFrame

We can also filter out the records containing the missing value entries from the PySpark DataFrame using the following code:

```
df.where(col('Sepallength').isNull()).show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|       null|       3.0|        1.6|       0.2| setosa|
|       null|       3.4|        1.6|       0.4| setosa|
+-----------+----------+-----------+----------+-------+
```

Figure 5.4: Iris DataFrame, fetching the missing value

The **show** function displays the first 20 records of a PySpark DataFrame. We only get two here as the **Sepallength** column only has two records with missing entries.

Handling Missing Values in Spark DataFrames

Missing value handling is one of the complex areas of data science. There are a variety of techniques that are used to handle missing values depending on the type of missing data and the business use case at hand.

These methods range from simple logic-based methods to advanced statistical methods such as regression and KNN. However, irrespective of the method used to tackle the missing values, we will end up performing one of the following two operations on the missing value data:

- Removing the records with missing values from the data
- Imputing the missing value entries with some constant value

In this section, we will explore how to do both these operations with PySpark DataFrames.

Exercise 40: Removing Records with Missing Values from a DataFrame

In this exercise, we will remove the records containing missing value entries for the PySpark DataFrame. Let's perform the following steps:

1. To remove the missing values from a particular column, use the following command:

   ```
   df.select('Sepallength').dropna().count()
   ```

   The previous code will return **148** as the output as the two records containing missing entries for the **Sepallength** column have been removed from the PySpark DataFrame.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. To remove all the records containing any missing value entry for any column from the PySpark DataFrame, use the following command:

```
df.dropna().count()
```

The DataFrame had **3** records with missing values, as we saw in *Exercise 2: Counting Missing Values in all DataFrame Columns*–two records with missing entries for the **Sepallength** column and one with a missing entry for the **Petallength** column.

The previous code removes all three records, thereby returning 147 complete records in the PySpark DataFrame.

Exercise 41: Filling Missing Values with a Constant in a DataFrame Column

In this exercise, we will replace the missing value entries of the PySpark DataFrame column with a constant numeric value.

Our DataFrame has missing values in two columns–**Sepallength** and **Petallength**:

1. Now, let's replace the missing value entries in both these columns with a constant numeric value of **1**:

```
y = df.select('Sepallength','Petallength').fillna(1)
```

2. Now, let's count the missing values in the new DataFrame, **y**, that we just created. The new DataFrame should have no missing values:

```
y.select([count(when(isnan(i) | col(i).isNull(), i)).alias(i) for i in
y.columns]).show()
```

The output is as follows:

```
+-----------+-----------+
|Sepallength|Petallength|
+-----------+-----------+
|          0|          0|
+-----------+-----------+
```

Figure 5.5: Iris DataFrame, finding the missing value

Sometimes, we want to replace all the missing values in the DataFrame with a single constant value.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Use the following command to replace all the missing values in the PySpark DataFrame with a constant numeric value of 1:

```
z = df.fillna(1)
```

4. Now, let's count the missing values in the new DataFrame **z** that we just created. The new DataFrame should have no missing values:

```
z.select([count(when(isnan(k) | col(k).isNull(), k)).alias(k) for k in
z.columns]).show()
```

The output is as follows:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|          0|         0|          0|         0|      0|
+-----------+----------+-----------+----------+-------+
```

Figure 5.6: Iris DataFrame, printing the missing value

Correlation

Correlation is a statistical measure of the level of association between two numerical variables. It gives us an idea of how closely two variables are related with each other. For example, age and income are quite closely related variables. It has been observed that the average income grows with age within a threshold. Thus, we can assume that age and income are positively correlated with each other.

> **Note**
>
> However, correlation does not establish a **cause-effect relationship**. A cause-effect relationship means that one variable is causing a change in another variable.

The most common metric used to compute this association is the **Pearson Product-Moment Correlation**, commonly known as **Pearson correlation coefficient** or simply as the **correlation coefficient**. It is named after its inventor, *Karl Pearson*.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The Pearson correlation coefficient is computed by dividing the covariance of the two variables by the product of their standard deviations. The correlation value lies between -1 and +1 with values close to 1 or -1 signifying strong association and values close to 0, signifying weak association. The sign (**+**, **-**) of the coefficient tells us whether the association is positive (both variables increase/decrease together) or negative (vice-versa).

> **Note**
>
> Correlation only captures the linear association between variables. So, if the association is non-linear, the correlation coefficient won't capture it. Two disassociated variables will have a low or zero correlation coefficient, but variables with zero/low correlation values are not necessarily disassociated.

Correlation is of great importance in statistical analysis, as it helps explain the data and sometimes highlights predictive relationships between variables. In this section, we will learn how to compute correlation between variables and compute a correlation matrix in PySpark.

Exercise 42: Computing Correlation

In this exercise, we will compute the value of the Pearson correlation coefficient between two numerical variables and a correlation matrix for all the numerical columns of our PySpark DataFrame. The correlation matrix helps us visualize the correlation of all the numerical columns with each other:

1. Perform the following steps to calculate the correlation between two variables:

   ```
   df.corr('Sepallength', 'Sepalwidth')
   ```

   The previous code outputs the Pearson correlation coefficient of **-0.1122503554120474** between the two variables mentioned.

2. Import the relevant modules, as illustrated here:

   ```
   from pyspark.mllib.stat import Statistics
   import pandas as pd
   ```

3. Remove any missing values from the data with the following command:

   ```
   z = df.fillna(1)
   ```

4. To remove any non-numerical columns before computing the correlation matrix, use the following command:

   ```
   a = z.drop('Species')
   ```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. Now, let's compute the correlation matrix with the help of the following command:

```
features = a.rdd.map(lambda row: row[0:])
correlation_matrix = Statistics.corr(features, method="pearson")
```

6. To convert the matrix into a pandas DataFrame for easy visualization, execute the following command:

```
correlation_df = pd.DataFrame(correlation_matrix)
```

7. Rename the indexes of the pandas DataFrame with the name of the columns from the original PySpark DataFrame:

```
correlation_df.index, correlation_df.columns = a.columns, a.columns
```

8. Now, visualize the pandas DataFrame with the following command:

```
correlation_df
```

	Sepallength	**Sepalwidth**	**Petallength**	**Petalwidth**
Sepallength	1.000000	-0.115616	0.792472	0.745084
Sepalwidth	-0.115616	1.000000	-0.427570	-0.366126
Petallength	0.792472	-0.427570	1.000000	0.962741
Petalwidth	0.745084	-0.366126	0.962741	1.000000

Figure 5.7: Iris DataFrame, computing correlation

Activity 12: Missing Value Handling and Correlation Analysis with PySpark DataFrames

In this activity, we will detect and handle missing values in the Iris dataset. We will also compute the correlation matrix and verify the variables showing strong correlations by plotting them with each other and fitting a linear line on the plot:

1. Perform the initial procedure of importing packages and libraries in the Jupyter notebook.

2. Set up **`SparkContext`** and **`SQLContext`**.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Read the data from the CSV file into a Spark object:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
|        5.0|       3.6|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 5.8: Iris DataFrame, reading data from the DataFrame

4. Fill in the missing values in the **Sepallength** column with its column mean.
5. Compute the correlation matrix for the dataset. Make sure to import the required modules.
6. Remove the **String** columns from the PySpark DataFrame and compute the correlation matrix in the Spark DataFrame.
7. Convert the correlation matrix into a pandas DataFrame:

	Sepallength	Sepalwidth	Petallength	Petalwidth
Sepallength	1.000000	-0.113841	0.861480	0.807310
Sepalwidth	-0.113841	1.000000	-0.427570	-0.366126
Petallength	0.861480	-0.427570	1.000000	0.962741
Petalwidth	0.807310	-0.366126	0.962741	1.000000

Figure 5.9: Iris DataFrame, converting the correlation matrix into a pandas DataFrame

8. Load the required modules and plotting data to plot the variable pairs showing strong positive correlation and fit a linear line on them.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

This is the graph for `x = "Sepallength", y = "Petalwidth"`:

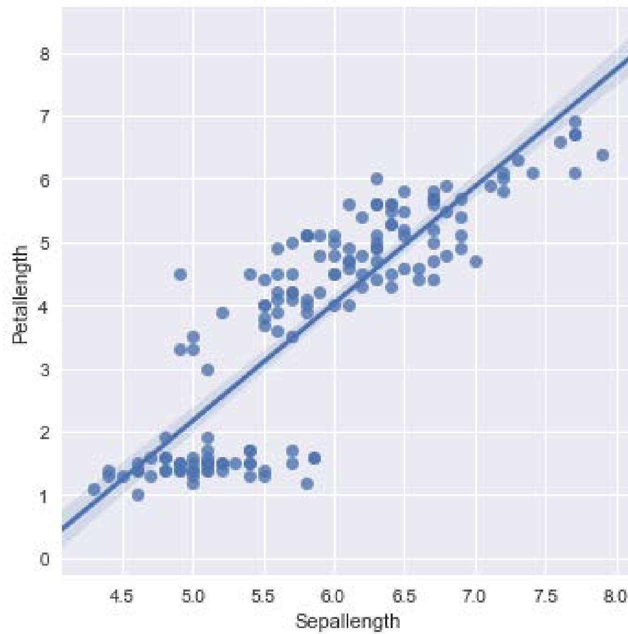


Figure 5.10: Iris DataFrame, plotting graph as x = "Sepallength", y = "Petalwidth"

This is the graph for `x = "Sepallength", y = "Petalwidth"`:

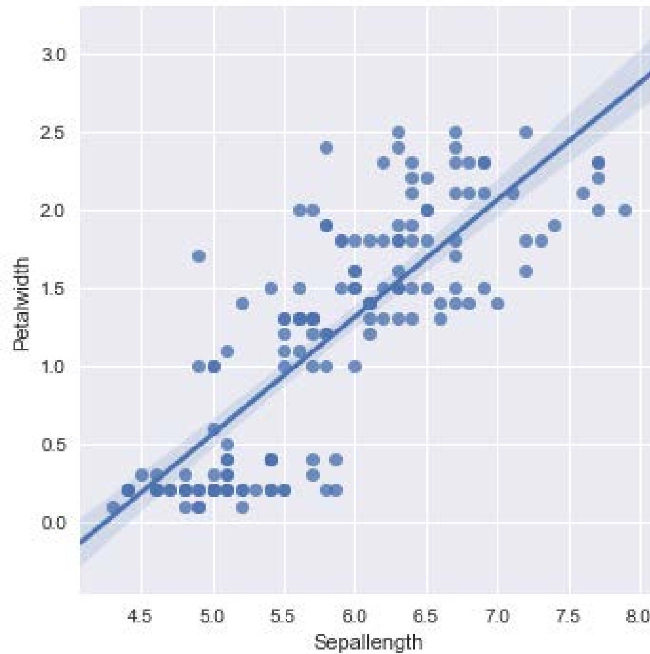


Figure 5.11: Iris DataFrame, plotting graph as x = "Sepallength", y = "Petalwidth"

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

This is the graph for `x = "Petallength", y = "Petalwidth"`:

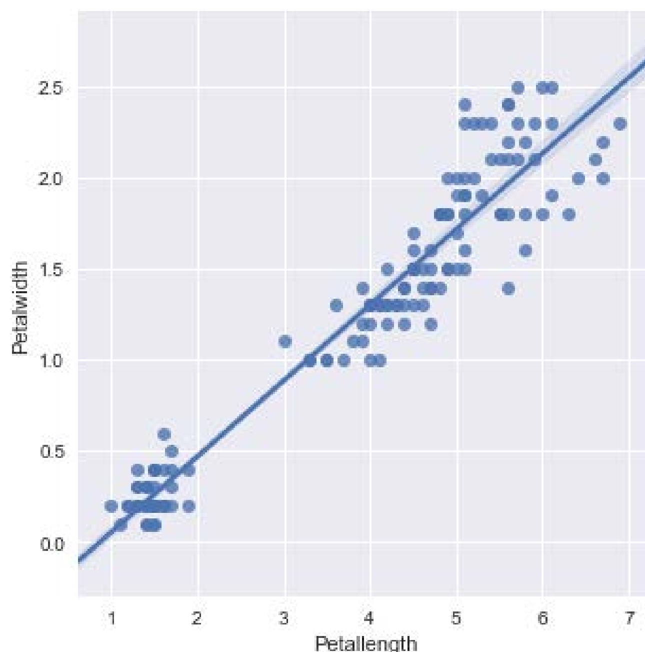


Figure 5.12: Iris DataFrame, plotting graph as x = "Petallength", y = "Petalwidth"

> **Note**
>
> Alternatively, you can use any dataset for this activity.
>
> The solution for this activity can be found on page 229.

Summary

In this chapter, we learned how to detect and handle the missing values in PySpark DataFrames. We looked at how to perform correlation and a metric to quantify the Pearson correlation coefficient. Later, we computed Pearson correlation coefficients for different numerical variable pairs and learned how to compute the correlation matrix for all the variables in the PySpark DataFrame.

In the next chapter, we will learn what problem definition is, and understand how to perform KPI generation. We will also use the data aggregation and data merge operations (learned about in previous chapters) and analyze data using graphs.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6

Exploratory Data Analysis

Learning Objectives

By the end of this chapter, you will be able to:

- Implement the concept of reproducibility with Jupyter notebooks
- Perform data gathering in a reproducible way
- Implement suitable code practices and standards to keep analysis reproducible
- Avoid the duplication of work by using IPython scripts

In this chapter, we will learn what problem definition is and how to use the KPI analysis techniques to enable coherent and well rounded analysis from the data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

One of the most important stages, and the initial step of a data science project, is understanding and defining a business problem. However, this cannot be a mere reiteration of the existing problem as a statement or a written report. To investigate a business problem in detail and define its purview, we can either use the existing business metrics to explain the patterns related to it or quantify and analyze the historical data and generate new metrics. Such identified metrics are the **Key Performance Indicators** (**KPIs**) that measure the problem at hand and convey to business stakeholders the impact of a problem.

This chapter is all about understanding and defining a business problem, identifying key metrics related to it, and using these identified and generated KPIs through pandas and similar libraries for descriptive analytics. The chapter also covers how to plan a data science project through a structured approach and methodology, concluding with how to represent a problem using graphical and visualization techniques.

Defining a Business Problem

A business problem in data science is a *long-term* or *short-term* challenge faced by a business entity that can prevent business goals being achieved and act as a constraint for growth and sustainability that can otherwise be prevented through an efficient data-driven decision system. Some typical data science business problems are predicting the demand for consumer products in the coming week, optimizing logistic operations for **third-party logistics** (**3PL**), and identifying fraudulent transactions in insurance claims.

Data science and machine learning are not magical technologies that can solve these business problems by just ingesting data into pre-built algorithms. They are complex in terms of the approach and design needed to create end-to-end analytics projects.

When a business needs such solutions, you may end up in a situation that forms a requirement gap if a clear understanding of the final objective is not set in place. A strong foundation to this starts with defining the business problem quantitatively and then carrying out scoping and solutions in line with the requirements.

The following are a few examples of common data science use cases that will provide an intuitive idea about common business problems faced by the industry today that are solved through data science and analytics:

- Inaccurate demand/revenue/sales forecasts
- Poor customer conversion, churn, and retention
- Fraud and pricing in the lending industry and insurance
- Ineffective customer and vendor/distributor scoring

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

- Ineffective recommendation systems for cross-sell/up-sell
- Unpredictable machine failures and maintenance
- Customer sentiment/emotion analysis through text data
- Non-automation of repetitive tasks that require unstructured data analytics

As we all know, in the last few years, industries have been changing tremendously, driven by the technology and innovation landscape. With the pace of evolving technology, successful businesses adapt to it, which leads to highly evolving and complex business challenges and problems. Understanding new business problems in such a dynamic environment is not a straightforward process. Though, case-by-case, business problems may change, as well as the approaches to them. However, the approach can be generalized to a large extent.

The following pointers are a broad step-by-step approach for defining and concluding a business problem, and in the following section, a detailed description of each step is provided:

1. Problem identification
2. Requirement gathering
3. Data pipeline and workflow
4. Identifying measurable metrics
5. Documentation and presentation

> **Note**
>
> The target variable, or the study variable, which is used as the attribute/variable/column in the dataset for studying the business problem, is also known as the **dependent variable** (**DV**), and all other attributes that are considered for the analysis are called **independent variables** (**IVs**).

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Problem Identification

Let's start with an example where an **Asset Management Company** (**AMC**) that has a strong hold on customer acquisition in their mutual funds domain, that is, targeting the right customers and onboarding them, is looking for higher customer retention to improve the average customer revenue and wallet share of their premium customers through data science-based solutions.

Here, the business problem is how to increase the revenue from the existing customers and increase their wallet share.

The problem statement is "*How do we improve the average customer revenue and increase the wallet share of premium customers through customer retention analytics*?" Summarizing the problem as stated will be the first step to defining a business problem.

Requirement Gathering

Once the problem is identified, have a point-by-point discourse with your client, which can include a **subject matter expert** (**SME**) or someone who is well-versed in the problem.

Endeavor to comprehend the issue from their perspective and make inquiries about the issue from various points of view, understand the requirements, and conclude how you can define the problem from the existing historical data.

Now and again, you will find that clients themselves can't comprehend the issue well. In such cases, you should work with your client to work out a definition of the issue that is satisfactory to both of you.

Data Pipeline and Workflow

After you have comprehended the issue in detail, the subsequent stage is to characterize and agree on quantifiable metrics for measuring the problem, that is, agreeing with the clients about the metrics to use for further analysis. In the long run, this will spare you a ton of issues.

These metrics can be identified with the existing system for tracking the performance of the business, or new metrics can be derived from historical data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

When you study the metrics for tracking the problem, the data for identifying and quantifying the problem may come from multiple data sources, databases, legacy systems, real-time data, and so on. The data scientist involved with this has to closely

work with the client's data management teams to extract and gather the required data and push it into analytical tools for further analysis. For this, there needs to be a strong pipeline for acquiring data. The acquired data is further analyzed to identify its important attributes and how they change over time in order to generate the KPIs. This is a crucial stage in client engagement and working in tandem with their teams goes a long way toward making the work easier.

Identifying Measurable Metrics

Once the required data is gathered through data pipelines, we can develop descriptive models to analyze historical data and generate insights into the business problem. **Descriptive models/analytics** is all about knowing *what has happened in the past* through time trend analysis and the density distribution of data analysis, among other things. For this, several attributes from the historical data need to be studied in order to gain insights into which of the data attributes are related to the current problem.

An example, as explained in the previous case, is an AMC that is looking for solutions to their specific business problem with customer retention. We'll look into identifying how we can generate KPIs to understand the problem with retention.

For this, historical data is mined to analyze the customer transaction patterns of previous investments and derive KPIs from them. A data scientist has to develop these KPIs based on their relevance and efficiency in explaining the variability of the problem, or in this case, the retention of customers.

Documentation and Presentation

The final step is documenting the identified KPIs, their significant trends, and how they can impact the business in the long run. In the previous case of customer retention, all these metrics—**length of relationship**, **average transaction frequency**, **churn rate**—can act as KPIs and be used to explain the problem quantitatively.

If we observe the trend in the churn rate, and let's say in this example, it has an increasing trend in the last few months, and if we graphically represent this, the client can easily understand the importance of having predictive churn analytics in place to identify the customer before they churn, and targeting stronger retention.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The potential of having a retention system in place needs to be presented to the client for which the documentation and graphical representation of the KPIs need to be carried out. In the previous case, the identified KPIs, along with their change in pattern, need to be documented and presented to the client.

Translating a Business Problem into Measurable Metrics and Exploratory Data Analysis (EDA)

If a specific business problem comes to us, we need to identify the KPIs that define that business problem and study the data related to it. Beyond generating KPIs related to the problem, looking into the trends and quantifying the problem through **Exploratory Data Analysis** (**EDA**) methods will be the next step.

The approach to explore KPIs is as follows:

- Data gathering
- Analysis of data generation
- KPI visualization
- Feature importance

Data Gathering

The data that is required for analyzing the problem is part of defining the business problem. However, the selection of attributes from the data will change according to the business problem. Consider the following examples:

- If it is a recommendation engine or churn analysis of customers, we need to look into historical purchases and **Know Your Customer** (**KYC**) data, among other data.
- If it is related to forecasting demand, we need to look into daily sales data.

It needs to be concluded that the required data can change from problem to problem.

Analysis of Data Generation

From the available data sources, the next step is to identify the metrics related to the defined problem. Apart from the preprocessing of data (refer to *Chapter 1*, *The Python Data Science Stack*, for details on data manipulation), at times, we need to manipulate the data to generate such metrics, or they can be directly available in the given data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

For example, let's say we are looking at supervised analysis, such as a **predictive maintenance problem** (a problem where predictive analytics is used to predict the condition of in-service equipment or machinery before it fails), where sensor- or computer-generated log data is used. Although log data is unstructured, we can identify which of the log files explain the failures of machinery and which do not. Unstructured data is without columns or rows. For example, it can be in XML or a similar format. Computer-generated log data is an example. Such data needs to be converted into columns and rows or make them structured, or label them, that is, provide column names for the data by converting the data into rows and columns.

Another example is identifying the churn of customers and predicting future customers who may churn in the coming periods, where we have transactional data on purchases with the features related to each purchase. Here, we need to manipulate the data and transform the current data in order to identify which customers have churned and which have not from all the purchase-related data.

To explain this better, in the raw data, there may be many rows of purchases for each customer, with the date of purchase, the units purchased, the price, and so on. All the purchases related to a customer need to be identified as a single row whether the customer has churned (churned means customers who discontinue using a product or service, also known as customer attrition) or not and with all the related information.

Here, we will be generating a KPI: **Churned** or **Not Churned** attributes for a customer, and similarly for all customers. The identified variable that defines the business problem is the target variable. A target variable is also known as the **response variable** or **dependent variable**. In *Exercise* XX, *Generate the Feature Importance for the Target Variable and Carry Out* EDA, in this chapter, it's captured and defined through the **churn** attribute.

KPI Visualization

To understand the trends and patterns in the KPIs, we need to represent them through interactive visualization techniques. We can use different methods, such as boxplot, time-trend graphs, density plots, scatter plots, pie charts, and heatmaps. We will learn more on this in *Exercise* XX, *Generate the Feature Importance for the Target Variable and Carry Out* EDA, in this chapter.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Feature Importance

Once the target variable is identified, the other attributes in the data and their importance to it in explaining the variability of the target variable need to be studied. For this, we use association, variance, and correlation methods to establish relations with the target variable of other variables (**explanatory** or **independent** variables).

There are multiple feature-importance methods and algorithms that can be used depending on the type of variables in the study, such as Pearson Correlation, Chi-Square tests, and algorithms based on Gini variable importance, decision trees, and Boruta, among others.

> **Note**
>
> The **target variable** or the **study variable**, which is used as the attribute/variable/column in the dataset for studying the business problem is also known as the **dependent variable** (**DV**), and all other attributes that are considered for the analysis are called **independent variables** (**IVs**).

In the following exercise, we will cover data gathering and analysis-data **(data that is generated by merging or combining multiple data sources to get one dataset for the analysis)** generation with KPI visualization, and then in the subsequent exercise, we will cover what feature importance is.

Exercise 43: Identify the Target Variable and Related KPIs from the Given Data for the Business Problem

Let's take the example of a **subscription problem** in the banking sector. We will use data that is from direct marketing campaigns by a Portuguese banking institution, where a customer either opens a term deposit (ref: https://www.canstar.com.au/term-deposits/what-is-a-term-deposit/) or not after the campaign. The subscription problem is characterized or defined contrastingly by every organization. For the most part, the customers who will subscribe to a service (here, it is a term deposit) have higher conversion potential (that is, from lead to customer conversion) to a service or product. Thus, in this problem, subscription metrics, that is, the outcome of historical data, is considered as the target variable or KPI.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

We will use descriptive analytics to explore trends in the data. We will start by identifying and defining the target variable (here, subscribed or not subscribed) and the related KPIs:

1. Download the bank.csv data from the following online resources:
   - https://archive.ics.uci.edu/ml/datasets/Bank+Marketing
   - https://archive.ics.uci.edu/ml/machine-learning-databases/00222/
2. Create a folder for the exercise (**packt_exercises**) and save the downloaded data there.
3. Start Jupyter notebook and import all the required libraries as illustrated. Now set the working directory using the **os.chdir()** function:

```
import numpy as np
import pandas as pd
import seaborn as sns
import time
import re
import os
import matplotlib.pyplot as plt
sns.set(style="ticks")

os.chdir("/Users/svk/Desktop/packt_exercises")
```

4. Use the following code to read the CSV and explore the dataset:

```
df = pd.read_csv('bank.csv', sep=';')
df.head(5)
print(df.shape)
df.head(5)
df.info()
df.describe()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

5. After executing the previous command, you will get output similar to the following:

	age	balance	day	duration	campaign	pdays	previous
count	4521.000000	4521.000000	4521.000000	4521.000000	4521.000000	4521.000000	4521.000000
mean	41.170095	1422.657819	15.915284	263.961292	2.793630	39.766645	0.542579
std	10.576211	3009.638142	8.247667	259.856633	3.109807	100.121124	1.693562
min	19.000000	-3313.000000	1.000000	4.000000	1.000000	-1.000000	0.000000
25%	33.000000	69.000000	9.000000	104.000000	1.000000	-1.000000	0.000000
50%	39.000000	444.000000	16.000000	185.000000	2.000000	-1.000000	0.000000
75%	49.000000	1480.000000	21.000000	329.000000	3.000000	-1.000000	0.000000
max	87.000000	71188.000000	31.000000	3025.000000	50.000000	871.000000	25.000000

Figure 6.1: Bank DataFrame

When studying the target variable (subscribed or not subscribed−**y**), it is important to look at the distribution of it. The type of target variable in this dataset is categorical, or of multiple classes. In this case, it's binary (Yes/No).

When the distribution is skewed to one class, the problem is known as an *imbalance in the variable*. We can study the proportion of the target variable using a bar plot. This gives us an idea about how many of each class there is (in this case, how many each of no and yes). The proportion of no is way higher than yes, which explains the imbalance in the data.

6. Let's execute the following commands to plot a bar plot for the given data:

```
count_number_susbc = df["y"].value_counts()
sns.barplot(count_number_susbc.index, count_number_susbc.values)

df['y'].value_counts()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
no     4000
yes     521
Name: y, dtype: int64
```

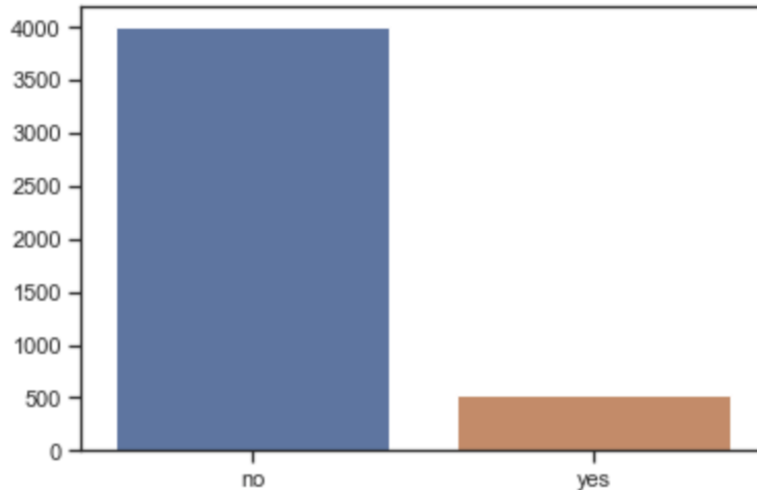


Figure 6.2: Bar plot

7. Now, we will take each variable and look at their distribution trends. The following histogram, which is provided as an example, is of the '`age`' column (attribute) in the dataset. Histograms/density plots are a great way to explore numerical/float variables, similar to bar plots. They can be used for categorical data variables. Here, we will show two numerical variables, `age` and `balance`, as examples using a histogram, and two categorical variables, `education` and `month`, using bar plots:

```
# histogram for age (using matplotlib)
plt.hist(df['age'], color = 'grey', edgecolor = 'black',
         bins = int(180/5))

# histogram for age (using seaborn)
sns.distplot(df['age'], hist=True, kde=False,
             bins=int(180/5), color = 'blue',
             hist_kws={'edgecolor':'black'})
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The histogram is as follows:

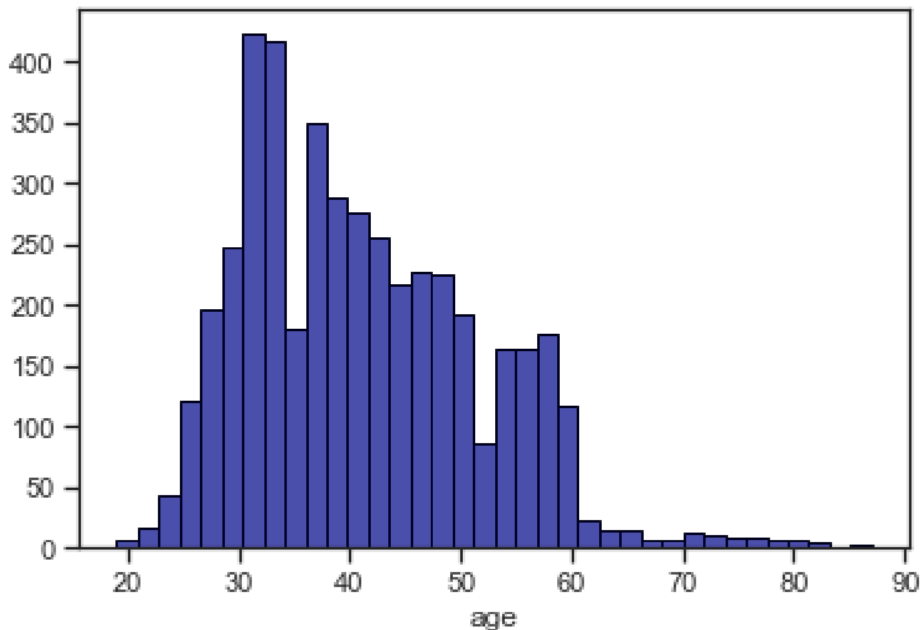


Figure 6.3: Histogram for age

8. To plot the histogram for the **balance** attribute in the dataset, use the following command:

```
# histogram for balance (using matplotlib)
plt.hist(df['balance'], color = 'grey', edgecolor = 'black',
         bins = int(180/5))

# histogram for balance (using seaborn)
sns.distplot(df['balance'], hist=True, kde=False,
             bins=int(180/5), color = 'blue',
             hist_kws={'edgecolor':'black'})
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The histogram for balance is as follows:

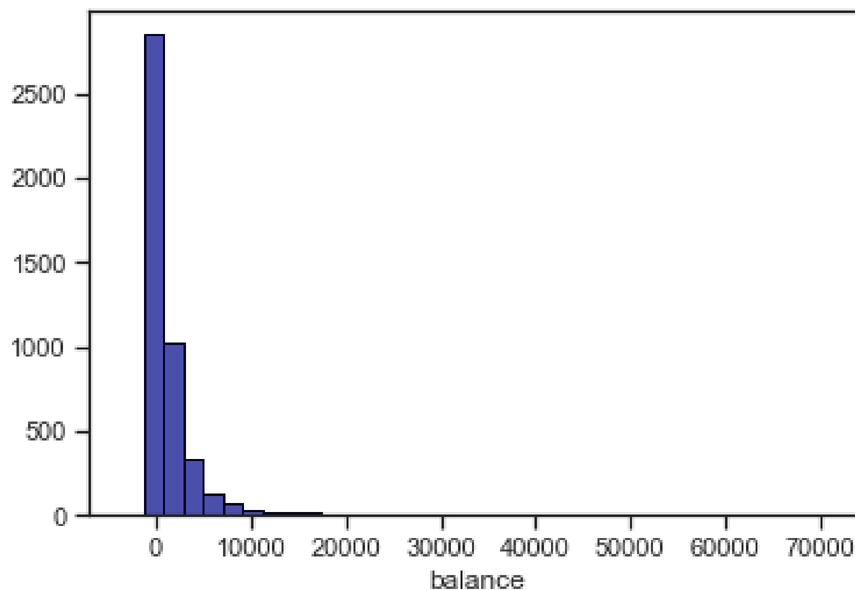


Figure 6.4: Histogram for balance

9. Now, using the following code, plot a bar plot for the **`education`** attribute in the dataset:

```
# barplot for the variable 'education'
count_number_susbc = df["education"].value_counts()
sns.barplot(count_number_susbc.index, count_number_susbc.values)

df['education'].value_counts()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The bar plot for the **education** attribute is as follows:

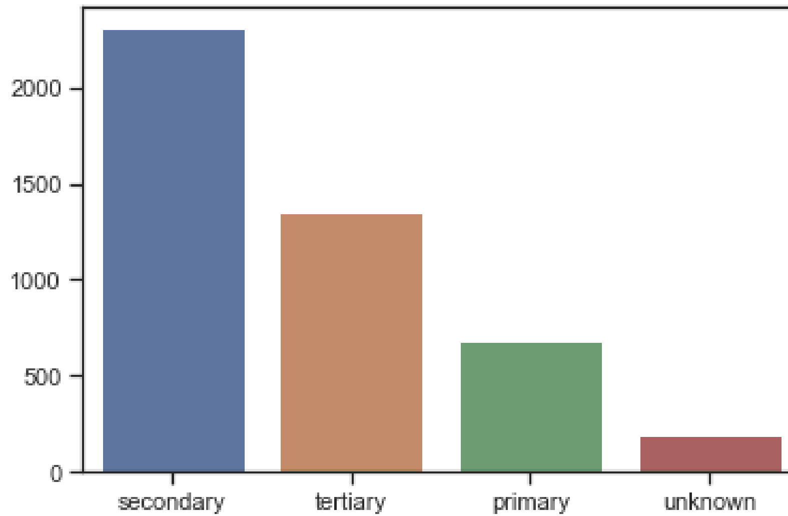


Figure 6.5: Bar plot for education

10. Use the following command to plot a bar plot for the **month** attribute of the dataset:

```
# barplot for the variable 'month'
count_number_susbc = df["month"].value_counts()
sns.barplot(count_number_susbc.index, count_number_susbc.values)

df['education'].value_counts()
```

The plotted graph is as follows:

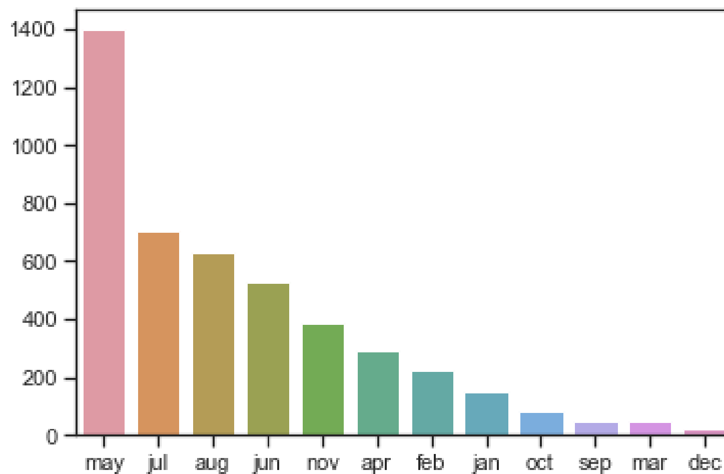


Figure 6.6: Bar plot for month

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

11. The next task is to generate a distribution for each class of the target variable to compare the distributions. Plot a histogram of the **age** attribute for the target variable (**yes**/**no**):

```
# generate separate list for each subscription type for age

x1 = list(df[df['y'] == 'yes']['age'])
x2 = list(df[df['y'] == 'no']['age'])

# assign colors for each subscription type
colors = ['#E69F00', '#56B4E9']
names = ['yes', 'no']

# plot the histogram
plt.hist([x1, x2], bins = int(180/15), density=True,
         color = colors, label=names)

# plot formatting
plt.legend()
plt.xlabel('IV')
plt.ylabel('prob distr (IV) for yes and no')
plt.title('Histogram for Yes and No Events w.r.t. IV')
```

The bar plot of the **month** attribute target variable is as follows:

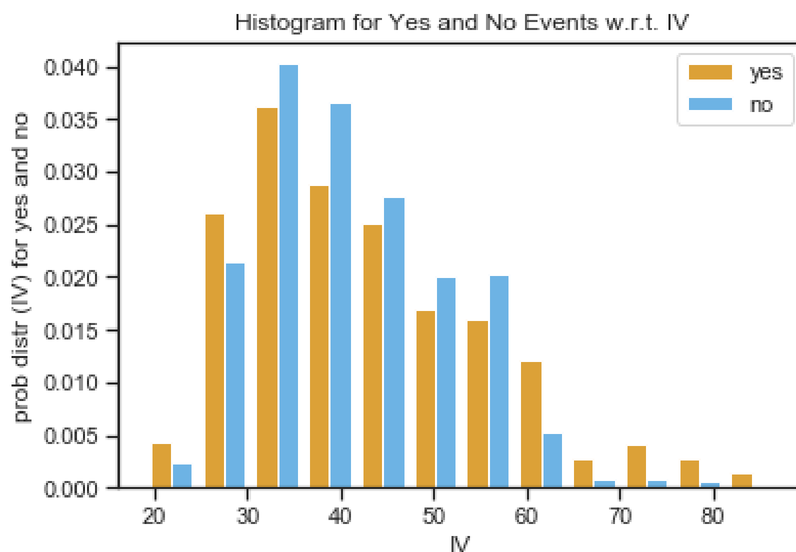


Figure 6.7: Bar plot of the month attribute for the target variable

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Now, using the following command, plot a bar plot for the target variable grouped by month:

```
df.groupby(["month", "y"]).size().unstack().plot(kind='bar', stacked=True,
figsize=(20,10))
```

The plotted graph is as follows:

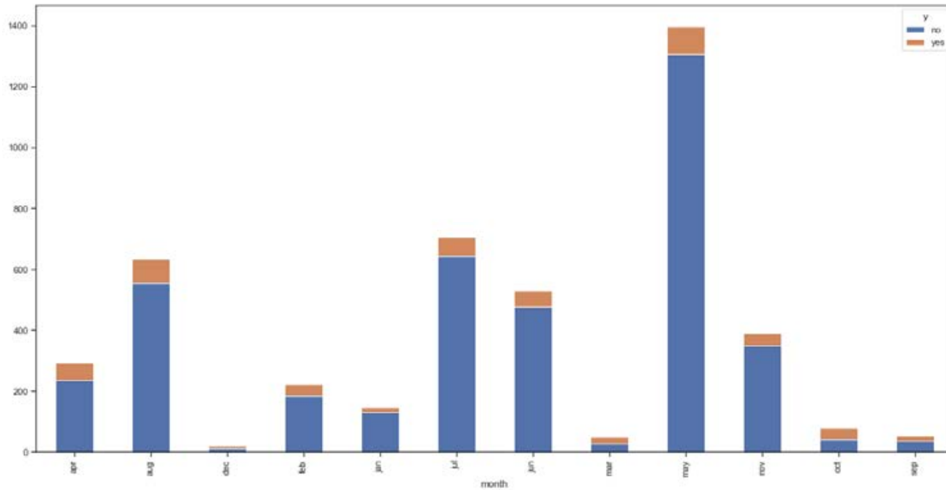


Figure 6.8: Histogram grouped by month

In this exercise, we looked into establishing KPIs and the target variable—**data gathering** and **analysis-data (data that is generated by merging or combining multiple data sources to get one dataset for analysis) generation**. The KPIs and target variable have been determined—**KPI visualization**. Now, in the next exercise, we will identify which of the variables are important in terms of explaining the variance of the target variable—**feature importance**.

Exercise 44: Generate the Feature Importance of the Target Variable and Carry Out EDA

In the previous exercise, we looked into the trends of the attributes, identifying their distribution, and how we can use various plots and visualization methods to carry these out. Prior to tackling a modeling problem, whether it is a predictive or a classification problem (for example, from the previous marketing campaign data, how to predict future customers that will have the highest probability of converting), we have to pre-process the data and select the important features that will impact the subscription campaign output models. To do this, we have to see the association of the attributes to the outcome (the target variable), that is, how much variability of the target variable is explained by each variable.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Associations between variables can be drawn using multiple methods; however, we have to consider the data type when choosing a method/algorithm. For example, if we are studying numerical variables (integers that are ordered, floats, and so on), we can use correlation analysis; if we are studying categorical variables with multiple classes, we can use Chi-Square methods. However, there are many algorithms that can handle both together and provide measurable outcomes to compare the importance of variables.

In this exercise, we will look at how various methods can be used to identify the importance of features:

1. Download the **bank.csv** file and read the data using the following command:

```
import numpy as np
import pandas as pd
import seaborn as sns
import time
import re
import os
import matplotlib.pyplot as plt
sns.set(style="ticks")

# set the working directory # in the example, the folder
# 'packt_exercises' is in the desktop
os.chdir("/Users/svk/Desktop/packt_exercises")

# read the downloaded input data (marketing data)
df = pd.read_csv('bank.csv', sep=';')
```

2. Develop a correlation matrix using the following command to identify the

   correlation between the variables:

```
df['y'].replace(['yes','no'],[1,0],inplace=True)
df['default'].replace(['yes','no'],[1,0],inplace=True)
df['housing'].replace(['yes','no'],[1,0],inplace=True)
df['loan'].replace(['yes','no'],[1,0],inplace=True)
corr_df = df.corr()
sns.heatmap(corr_df, xticklabels=corr_df.columns.values, yticklabels=corr_
df.columns.values, annot = True, annot_kws={'size':12})
heat_map=plt.gcf(); heat_map.set_size_inches(10,5)
plt.xticks(fontsize=10); plt.yticks(fontsize=10); plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The plotted correlation matrix heatmap is as follows:

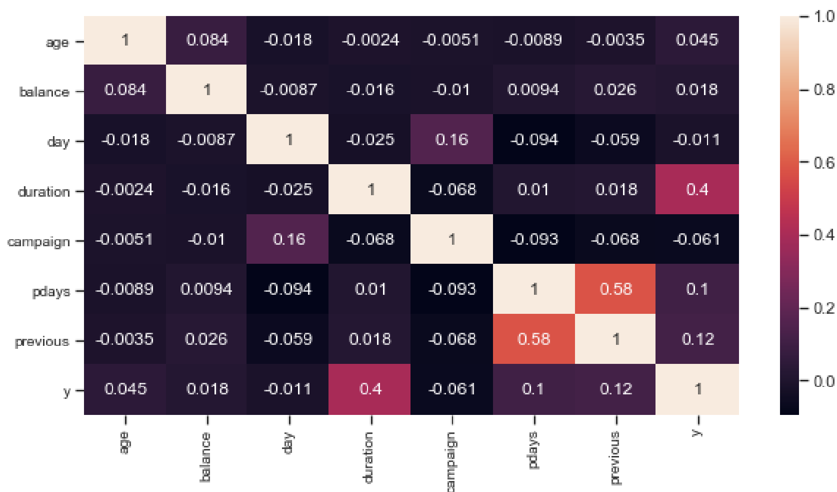


Figure 6.9: Correlation matrix

> **Note**
>
> **Correlation analysis** is carried out to analyze the relationships between numerical variables. Here, we converted the *categorical target* variable into *binary values* and looked into how it correlates with other variables.
>
> The correlation coefficient values in the matrix can range from **-1** to **+1**, where a value close to **0** signifies no relation, -1 signifies a relation where one variable reduces as the other increases (inverse), and +1 signifies that as one variable increases the other also increases (direct).
>
> High correlation among independent variables (which are all variables except the target variable) can lead to multicollinearity among the variables, which can affect the predictive model accuracy.

> **Note**
>
> Make sure to install boruta if not installed already using the following command:

```
pip install boruta --upgrade
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Build a feature importance output based on Boruta (a wrapper algorithm around a random forest):

```
# import DecisionTreeClassifier from sklearn and
# BorutaPy from boruta

import numpy as np
from sklearn.ensemble import RandomForestClassifier
from boruta import BorutaPy

# transform all categorical data types to integers (hot-encoding)
for col_name in df.columns:
    if(df[col_name].dtype == 'object'):
        df[col_name]= df[col_name].astype('category')
        df[col_name] = df[col_name].cat.codes

# generate separate dataframes for IVs and DV (target variable)
X = df.drop(['y'], axis=1).values
Y = df['y'].values


# build RandomForestClassifier, Boruta models and
# related parameter
rfc = RandomForestClassifier(n_estimators=200, n_jobs=4, class_
weight='balanced', max_depth=6)
boruta_selector = BorutaPy(rfc, n_estimators='auto', verbose=2)
n_train = len(X)

# fit Boruta algorithm
boruta_selector.fit(X, Y)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
Iteration:      1 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      2 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      3 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      4 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      5 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      6 / 100
Confirmed:      0
Tentative:      16
Rejected:       0
Iteration:      7 / 100
```

Figure 6.10: Fit Boruta algorithm

4. Check the rank of features as follows:

```
feature_df = pd.DataFrame(df.drop(['y'], axis=1).columns.tolist(), 
columns=['features'])
feature_df['rank']=boruta_selector.ranking_
feature_df = feature_df.sort_values('rank', ascending=True).reset_
index(drop=True)
sns.barplot(x='rank',y='features',data=feature_df)
feature_df
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

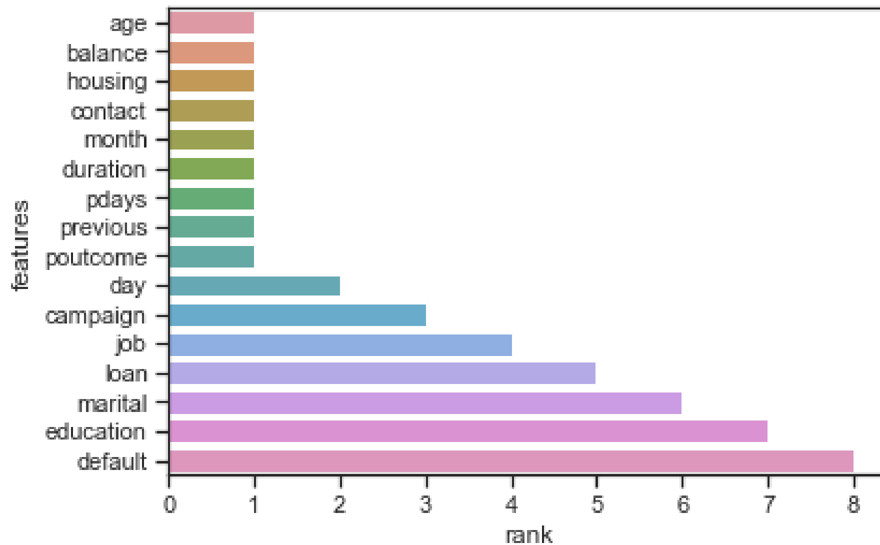


Figure 6.11: Boruta output

Structured Approach to the Data Science Project Life Cycle

Embarking on data science projects needs a robust methodology in planning the project, taking into consideration the potential scaling, maintenance, and team structure. We have learned how to define a business problem and quantify it with measurable parameters, so the next stage is a project plan that includes the development of the solution, to the deployment of a consumable business application.

This topic puts together some of the best industry practices structurally with examples for data science project life cycle management. This approach is an idealized sequence of stages; however, in real applications, the order can change according to the type of solution that is required.

Typically, a data science project for a single model deployment takes around three months, but this can increase to six months, or even up to a year. Defining a process from data to deployment is the key to reducing the time to deployment.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Data Science Project Life Cycle Phases

The stages of a data science project life cycle are as follows:

1. Understanding and defining the business problem
2. Data access and discovery
3. Data engineering and pre-processing
4. Modeling development and evaluation

Phase 1: Understanding and Defining the Business Problem

Every data science project starts with learning the business domain and framing the business problem. In most organizations, the application of advanced analytics and data science techniques is a fledgling discipline, and most of the data scientists involved will have a limited understanding of the business domains.

To understand the business problem and the domain, the key stakeholders and **subject matter experts** (**SMEs**) need to be identified. Then, the SMEs and the data scientists interact with each other to develop the initial hypothesis and identify the data sources required for the solution's development. This is the first phase of understanding a data science project.

Once we have a well-structured business problem and the required data is identified, along with the data sources, the next phase of data discovery can be initiated. The first phase is crucial in building a strong base for scoping and the solution approach.

Phase 2: Data Access and Discovery

This phase includes identifying the data sources and building data pipelines and data workflows in order to acquire the data. The nature of the solution and the dependent data can vary from one problem to another in terms of the structure, velocity, and volume.

At this stage, it's important to identify how to acquire the data from the data sources. This can be through direct connectors (libraries available on Python) using database access credentials, having APIs that provide access to the data, directly crawling data from web sources, or can even be data dumps provided in CSVs for initial prototype developments. Once a strong data pipeline and workflow for acquiring data is established, the data scientists can start exploring the data to prepare the analysis-data **(data that is generated by merging or combining multiple data sources to get one dataset for analysis).**

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Phase 3: Data Engineering and Pre-processing

The pre-processing of data means transforming raw data into a form that's consumable by a machine learning algorithm. It is essentially the manipulation of data into a structure that's suitable for further analysis or getting the data into a form that can be input data for modeling purposes. Often, the data required for analysis can reside in several tables, databases, or even external data sources.

A data scientist needs to identify the required attributes from these data sources and merge the available data tables to get what is required for the analytics model. This is a tedious and time-consuming phase on which data scientists spend a considerable part of the development cycle.

The pre-processing of data includes activities such as outlier treatment, missing value imputation, scaling features, mapping the data to a Gaussian (or normal) distribution, encoding categorical data, and discretization, among others.

To develop robust machine learning models, it's important that data is pre-processed efficiently.

> **Note**
>
> Python has a few libraries that are used for data pre-processing. scikit-learn has many efficient built-in methods for pre-processing data. The scikit-learn documentation can be found at https://scikit-learn.org/stable/modules/preprocessing.html.

Let's go through the following activity to understand how to carry out data engineering and pre-processing using one of the pre-processing techniques, such as Gaussian normalization.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 13: Carry Out Mapping to Gaussian Distribution of Numeric Features from the Given Data

Various pre-processing techniques need to be carried out to prepare the data before pushing it into an algorithm.

> **Note**
>
> Explore this website to find various methods for pre-processing: https://scikit-learn.org/stable/modules/preprocessing.html.

In this exercise, we will carry out data normalization, which is important for parametric models such as linear regression, logistic regression, and many others:

1. Use the **`bank.csv`** file and import all the required packages and libraries into the Jupyter notebook.
2. Now, determine the numeric data from the DataFrame. Categorize the data according to its type, such as categorical, numeric (float or integer), date, and so on. Let's carry out normalization on numeric data.
3. Carry out a normality test and identify the features that have a non-normal distribution.
4. Plot the probability density of the features to visually analyze the distribution of the features.
5. Prepare the power transformation model and carry out transformations on the identified features to convert them to normal distribution based on the **`box-cox`** or **`yeo-johnson`** method.
6. Apply the transformations on new data (evaluation data) with the same generated parameters for the features that were identified in the training data.

> **Note**
>
> Multiple variables' density plots are shown in the previous plot after the transformations are carried out. The distribution of the features in the plots is closer to a Gaussian distribution.
>
> The solution for this activity can be found on page 236.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Once the transformations are carried out, we analyze the normality of the features again to see the effect. You can see that after the transformations, some of the features can have the null hypothesis not rejected (that is, the distribution is still not Gaussian) with almost all features higher **p** values (refer to point 2 of this activity). Once the transformed data is generated, we bind it back to the numeric data.

Phase 4: Model Development

Once we have a cleaned and pre-processed data, it can be ingested into a machine learning algorithm for exploratory or predictive analysis, or for other applications. Although the approach to a problem may be designed, whether it's a classification, association, or a regression problem, for example, the specific algorithm that needs to considered for the data has to be identified. For example, if it's a classification problem, it could be a decision tree, a support vector machine, or a neural network with many layers.

For modeling purposes, the data needs to be separated into testing and training data. The model is developed on the training data and its performance (accuracy/error) is evaluated on the testing data. With the algorithms selected, the related parameters with respect to it need to be tuned by the data scientist to develop a robust model.

Summary

In this chapter, we learned how to define a business problem from a data science perspective through a well-defined, structured approach. We started by understanding how to approach a business problem, how to gather the requirements from stakeholders and business experts, and how to define the business problem by developing an initial hypothesis.

Once the business problem was defined with data pipelines and workflows, we looked into understanding how to start the analysis on the gathered data in order to generate the KPIs and carry out descriptive analytics to identify the key trends and patterns in the historical data through various visualization techniques.

We also learned how a data science project life cycle is structured, from defining the business problem to various pre-processing techniques and model development. In the next chapter, we will be learning how to implement the concept of high reproducibility on a Jupyter notebook, and its importance in development.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7

Reproducibility in Big Data Analysis

Learning Objectives

By the end of this chapter, you will be able to:

- Implement the concept of reproducibility with Jupyter notebooks
- Perform data gathering in a reproducible way
- Implement suitable code practices and standards to keep analysis reproducible
- Avoid the duplication of work with IPython scripts

In this chapter, we will discover how reproducibility plays a vital role in big data analysis.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

In the previous chapter, we learned how to define a business problem from a data science perspective through a very structured approach, which included how to identify and understand business requirements, an approach to solutioning it, and how to build data pipelines and carry out analysis.

In this chapter, we will look at the reproducibility of computational work and research practices, which is a major challenge faced today across the industry, as well as by academics—especially in data science work, in which most of the data, complete datasets, and associated workflow cannot be accessed completely.

Today, most research and technical papers conclude with the approach used on the sample data, a brief mention of the methodology used, and a theoretical approach to a solution. Most of these works lack detailed calculations and step-by-step approaches. This is a very limited amount of knowledge for anyone reading it to be able to reproduce the same work that was carried out. This is the basic objective of reproducible coding, where ease of reproducing the code is key.

There have been advancements in notebooks in general, which can include text elements for commenting in detail; this improves the reproduction process. This is where Jupyter as a notebook is gaining traction within the data science and research communities.

Jupyter was developed with the intention of being open source software with open standards and services for interactive computing across dozens of programming languages, including Python, Spark, and R.

Reproducibility with Jupyter Notebooks

Let's start by learning what it is meant by **computational reproducibility**. Research, solutions, prototypes, and even a simple algorithm that is developed is said to be reproducible if access is provided to the original source code that was used to develop the solution, and the data that was used to build any related software should be able to produce the same results. However, today, the scientific community is experiencing some challenges in reproducing work developed previously by peers. This is mainly due to the lack of documentation and difficulty in understanding process workflows.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The impact of a lack of documentation can be seen at every level, right from understanding the approach to the code level. Jupyter is one of the best tools for improvising this process, for better reproducibility, and for the reuse of developed code. This includes not just understanding what each line or snippet of code does, but also understanding and visualizing data.

> **Note**
>
> Jon Claerbout, who is considered the father of reproducible computational research, in the early 1990s required his students to develop research work and create results that could be regenerated in a single click. He believed that work that was completed took effort and tim, and should be left as it was so that further work on it could be done by reusing the earlier work without any difficulties. On a macro level, an economy's growth is strongly determined by the amount of innovation. The reproducibility of earlier work or research contributes to overall innovation growth.

Now let's see how we can maintain effective computational reproducibility using the Jupyter notebook.

The following pointers are broad ways to achieve reproducibility using a Jupyter notebook in Python:

- Provide a detailed introduction to the business problem
- Document the approach and workflow
- Explain the data pipelines
- Explain the dependencies
- Use source code version control
- Modularize the process

In the following sections, let's explore and discuss the previously mentioned topics in a brief manner.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction to the Business Problem

One of the key advantages of using the Jupyter notebook is that it includes textual content along with the code to create a workflow.

We must start with a good introduction to the business problem we have identified, and a summarization of it has to be documented in the Jupyter notebook to provide the gist of the problem. It has to include a problem statement, with the identified business problem from a data science perspective, concluding why we need to carry out this analysis, or what the objective of the process is.

Documenting the Approach and Workflows

In data science, there can be a lot of back and forth in computational work, such as, for instance, the explorations that are carried out, the kind of algorithms used, and the parameter changes to tune.

Once changes to the approach have been finalized, those changes need to be documented to avoid work being repeated. Documenting the approach and workflows helps to set up a process. Adding comments to code while developing is a must and this has to be a continuous practice rather than waiting until the end or the results to add comments. By the end of the process, you may have forgotten the details, and this could result in miscalculating the effort that goes into it. The advantages of maintaining the Jupyter notebook with good documentation are the following:

- Tracking the development effort
- Self-explanatory code with comments for each process
- A better understanding of the code workflow and the results of each step
- Avoiding back-and-forth work by making previous code snippets for specific tasks easy to find
- Avoiding duplicating work by understanding the repeated use of code
- Ease of knowledge transfer

Explaining the Data Pipeline

The data for identifying and quantifying the problem can be generated from multiple data sources, databases, legacy systems, real-time data sources, and so on. The data scientist involved in this closely works with the data management teams of the client to extract and gather the required data and ingest it into the analytical tools for further analysis, and creates a strong data pipeline to acquire this data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

It is important to document the data sources in detail (covered in the previous chapter) to maintain a data dictionary that explains the variables that are considered, why they are considered, what kind of data we have (structured or unstructured), and the type of data that we have; that is, whether we have a time-series, multivariate, or data that needs to be preprocessed and generated from raw sources such as image, text, speech, and so on.

Explain the Dependencies

Dependencies are the packages and libraries that are available in a tool. For instance, you may use OpenCV (https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html), a library in Python for image-related modeling, or you may use an API such as TensorFlow for deep-learning modeling. Here's another example: if you use Matplotlib (https://matplotlib.org/) for visualization in Python, Matplotlib can be a part of dependencies. On other hand, dependencies can include the hardware and software specifications that are required for an analysis. You can manage your dependencies explicitly from the beginning by employing a tool such as a Conda environment to list all relevant dependencies (covered in previous chapters on dependencies for pandas, NumPy, and so on), including their package/library versions.

Using Source Code Version Control

Version control is an important aspect when it comes to any kind of computational activity that involves code. When code is being developed, bugs or errors will arise. If previous versions of the code are available, we will then be able to pinpoint when the bug was identified, when it was resolved, and the amount of effort that went into it. This is possible through version control. At times, you may need to revert to older versions, because of scalability, performance, or for some other reasons. Using source code version control tools, you can always easily access previous versions of code.

Modularizing the Process

Avoiding duplicate code is an effective practice for managing repetitive tasks, for maintaining code, and for debugging. To carry this out efficiently, you must modularize the process.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's understand this in detail. Say you carry out a set of data manipulation processes, where you develop the code to complete a task. Now, suppose you need to use the same code in a later section of the code; you need to do add, copy, or run the same steps again, which is a repetitive task. The input data and variable names can change. To handle this, you can write the earlier steps as a function for a dataset or on a variable and save all such functions as a separate module. You can call it a functions file (for example, `functions.py`, a Python file).

In the next section, we will look at this in more detail, particularly with respect to gathering and building an efficient data pipeline in a reproducible way.

Gathering Data in a Reproducible Way

Once the problem is defined, the first step in an analysis task is gathering data. Data can be extracted from multiple sources: databases, legacy systems, real-time data, external data, and so on. Data sources and the way data can be ingested into the model needs to be documented.

Let's understand how to use markdown and code block functionalities in the Jupyter notebook. Text can be added to Jupyter notebooks using markdown cells. These texts can be changed to bold or italic, like in any text editor. To change the cell type to markdown, you can use the **Cell** menu. We will look at the ways you can use various functionalities in markdown and code cells in Jupyter.

Functionalities in Markdown and Code Cells

- **Markdown in Jupyter**: To select the markdown option in **Jupyter**, click on **Widgets** and **Markdown** from the drop-down menu:

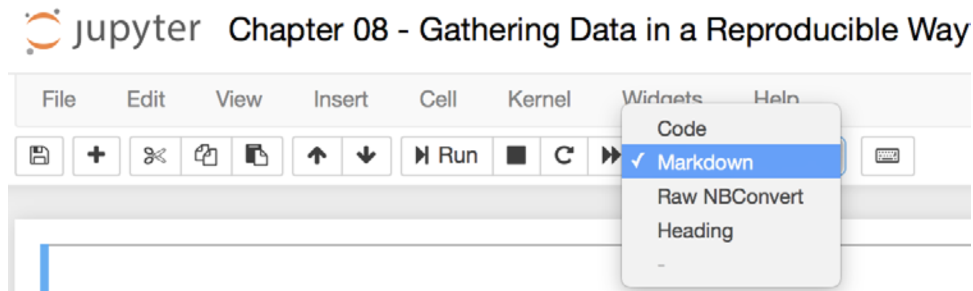


Figure 7.1: The markdown option in the Jupyter notebook

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

- **Heading in Jupyter**: There are two types of headings in the Jupyter notebook. The syntax to add a heading is similar to the way it is in HTML, using **<h1>** and **<h2>** tags:

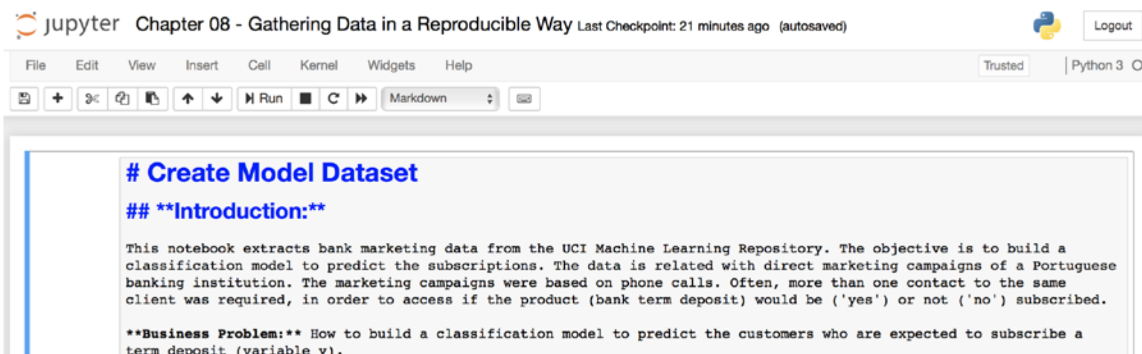


Figure 7.2: Heading levels in the Jupyter notebook

- **Text in Jupyter**: To add text just the way it is, we do not add any tags to it:

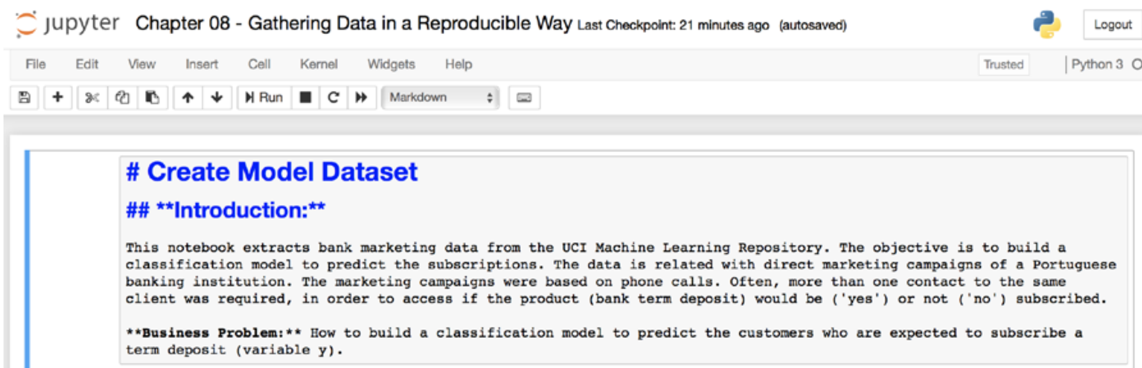


Figure 7.3: Using normal text in the Jupyter notebook

- **Bold in Jupyter**: To make text appear bold in the Jupyter notebook, add two stars (**) to the start and end of the text, for example, **Bold**:

```
**Business Problem:** How to build a classification model to predict the customers who are expected to subscribe a
term deposit (variable y).
```

Figure 7.4: Using bold text in the Jupyter notebook

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

- **Italic in Jupyter**: To make text appear in italics in the Jupyter notebook, add one star (*) to the start and end of the text:

```
*Business Problem:* How to build a classification model to predict the customers who are expected to subscribe a term
deposit (variable y).
```

Figure 7.5: Using italicized text in the Jupyter notebook

- **Code in Jupyter**: To make text appear as code, select the **Code** option from the dropdown:

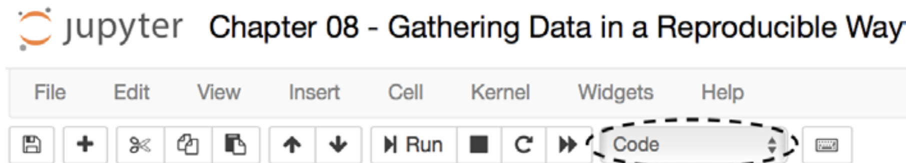


Figure 7.6: Code in the Jupyter notebook

Explaining the Business Problem in the Markdown

Provide a brief introduction to the business problem to understand the objective of the project. The business problem definition is a summarization of the problem statement and includes the way in which the problem can be resolved using a data science algorithm:

Introduction:

This notebook extracts bank marketing data from the UCI Machine Learning Repository. The objective is to build a classification model to predict the subscriptions. The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed.

Business Problem: How to build a classification model to predict the customers who are expected to subscribe a term deposit (variable y).

Figure 7.7: Snippet of the problem definition

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Providing a Detailed Introduction to the Data Source

The data source needs to be documented properly to understand the data license for reproducibility and for further work. A sample of how the data source can be added is as follows:

Data Source

Read the input data that is downloaded from the UCI Machine Library repository for Bank Marketing Data from the link:
https://archive.ics.uci.edu/ml/datasets/bank+marketing

```
# read the input dataset as 'df' using pandas' read_csv function
df = pd.read_csv('bank.csv', sep=';')

# view the first 5 rows of the dataset using head function
df.head(5)
```

Figure 7.8: Data Source in Jupyter notebook

Explain the Data Attributes in the Markdown

A data dictionary needs to be maintained to understand the data on an attribute level. This can include defining the attribute with what type of data it is:

Data Dictionary

Provides detailed attribute level information:

1 - age (numeric)

2 - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown')

3 - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)

4 - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')

5 - default: has credit in default? (categorical: 'no','yes','unknown')

6 - housing: has housing loan? (categorical: 'no','yes','unknown')

Figure 7.9: Detailed attributes in markdown

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

To understand the data on an attribute level, we can use functions such as **info** and **describe**; however, **pandas_profiling** is a library that provides a lot of descriptive information in one function, from which we can extract the following information:

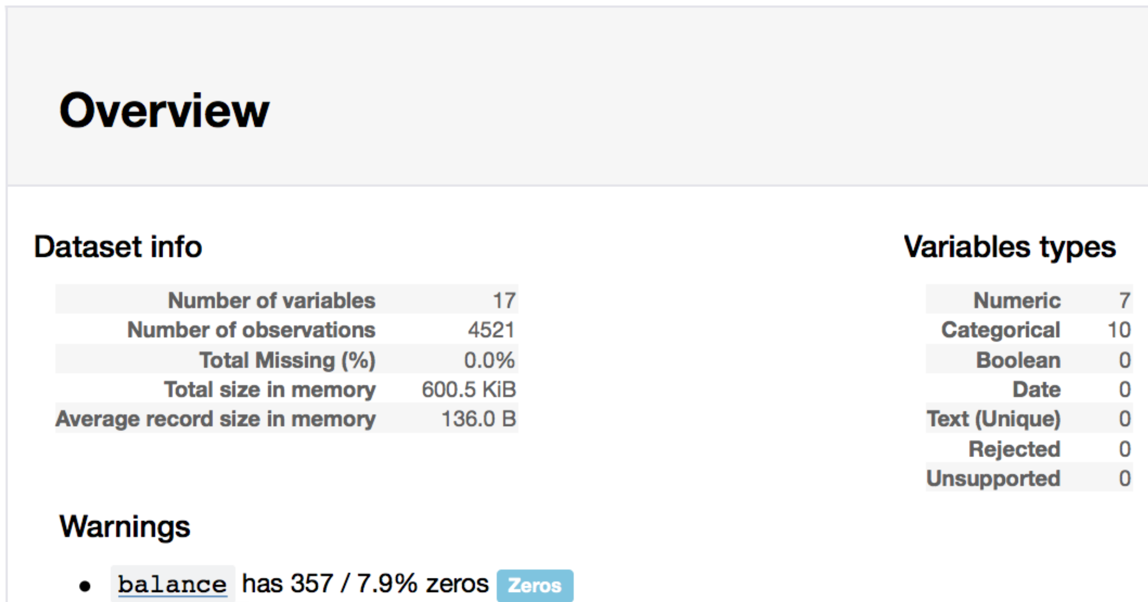


Figure 7.10: Profiling report

On the DataFrame level, that is, for the overall data, which includes all the columns and rows that are considered:

- Number of variables
- Number of observations
- Total missing (%)
- Total size in memory
- Average record size in memory
- Correlation matrix
- Sample data

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

On the attribute level, which is for a specific column, the specifications are as follows:

- Distinct count
- Unique (%)
- Missing (%)
- Missing (n)
- Infinite (%)
- Infinite (n)
- Histogram for distribution
- Extreme values

Figure 7.11: Data profiling report on the attribute level

Exercise 45: Performing Data Reproducibility

The aim of this exercise is to learn how to develop code for high reproducibility in terms of data understanding. We will be using the UCI Bank and Marketing dataset taken from this link: https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson07/Dataset/bank/bank.csv.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Let's perform the following steps to achieve data reproducibility:

1. Add headings and mention the business problem in the notebook using markup:

Create Model Dataset

Introduction:

This notebook extracts bank marketing data from the UCI Machine Learning Repository. The objective is to build a classification model to predict the subscriptions. The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed.

Business Problem: How to build a classification model to predict the customers who are expected to subscribe a term deposit (variable y).

Figure 7.12: Introduction and business problem

2. Import the required libraries into the Jupyter notebook:

```
import numpy as np
import pandas as pd
import time
import re
import os
import pandas_profiling
```

3. Now set the working directory, as illustrated in the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

4. Import and read the input dataset as **df** using pandas' **read_csv** function from the dataset:

```
df = pd.read_csv('bank.csv', sep=';')
```

5. Now view the first five rows of the dataset using the **head** function:

```
df.head(5)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	may	226	1	-1

Figure 7.13: Data in the CSV file

6. Add the **Data Dictionary** and **Data Understanding** sections in the Jupyter notebook:

Data Dictionary

Provides detailed attribute level information:

1 - age (numeric)

2 - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown')

3 - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)

4 - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')

5 - default: has credit in default? (categorical: 'no','yes','unknown')

6 - housing: has housing loan? (categorical: 'no','yes','unknown')

Figure 7.14: Data Dictionary

The **Data Understanding** section is as follows:

Data Understanding ¶

To understand the data at a attribute level, we can use functions like info and describe, however, pandas_profiling is a library that provides many descriptive information in one function where we can extract the following information:

At dataset level:

1. Number of variables
2. Number of observations
3. Total Missing (%)
4. Total size in memory
5. Average record size in memory
6. Correlation Matrix
7. Sample Data

Figure 7.15: Data Understanding

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. To understand the data specifications, use pandas profiling to generate the descriptive information:

```
pandas_profiling.ProfileReport(df)
```

The output is as follows:

Overview

Dataset info

Number of variables	17
Number of observations	4521
Total Missing (%)	0.0%
Total size in memory	600.5 KiB
Average record size in memory	136.0 B

Variables types

Numeric	7
Categorical	10
Boolean	0
Date	0
Text (Unique)	0
Rejected	0
Unsupported	0

Warnings

- `balance` has 357 / 7.9% zeros Zeros
- `previous` has 3705 / 82.0% zeros Zeros

Figure 7.16: Summary of the specifications with respect to data

> **Instructor Note:**
>
> This exercise identifies how a Jupyter notebook is created and includes how to develop a reproducible Jupyter notebook for a bank marketing problem. This must include a good introduction to the business problem, data, data types, data sources, and so on.

Code Practices and Standards

Writing code with a set of practices and standards is important for code reproducibility, as is explaining the workflow of the process descriptively in a step-wise manner.

This is universally applicable across any coding tool that you may use, not just with Jupyter. Some coding practices and standards should be followed strictly and a few of these will be discussed in the next section.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Environment Documentation

For installation purposes, you should maintain a snippet of code to install the necessary packages and libraries. The following practices help with code reproducibility:

- Include the versions used for libraries/packages.
- Download the original version of packages/libraries used and call the packages internally for installation in a new setup.
- Effective implementation by running it in a script that automatically installs dependencies.

Writing Readable Code with Comments

Code commenting is an important aspect. Apart from the markdown option available on Jupyter, we must include comments for each code snippet. At times, we make changes in the code in a way that may not be used immediately but will be required for later steps. For instance, we can create an object that may not be used immediately for the next step but for later steps. This can confuse a new user in terms of understanding the flow. Commenting such specifics is crucial.

When we use a specific method, we must provide a reason for using that particular method. For example, let's say, for the transformation of data for normal distribution, you can use **box-cox** or **yeo-johnson**. If there are negative values, you may prefer **yeo-johnson**, as it can handle negative values. It needs to be commented as in the following example:

```
# create a PowerTransformer based transformation (box-cox) (note: box-cox can handle only positive values)
#pt = preprocessing.PowerTransformer(method='box-cox', standardize=False) # applicable if box-cox is used
pt = preprocessing.PowerTransformer(method='yeo-johnson', standardize=True, copy=True) # applicable if yeo-johnson is
```

Figure 7.17: Comments with reasons

We should also follow a good practice for naming the objects that we create. For example, you can name raw data **raw_data**, and can do the same for model data, preprocessed data, analysis data, and so on. The same goes when creating objects such as models and methods, for example, we can call power transformations **pt**.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Effective Segmentation of Workflows

When developing code, there are steps that you design to achieve end results. Each step can be part of a process. For instance, reading data, understanding data, carrying out various transformations, or building a model. Each of these steps, needs to be clearly separated for multiple reasons; firstly, code readability for how each stage is carried out and, secondly, how the result is generated at each stage.

For example, here, we are looking at two sets of activities. One where the loop is generated to identify the columns that need to be normalized, and the second generating the columns that do not need to be normalized using the previous output:

```
# loop for identifying the columns with non-normal distribution on the transformed data
numeric_df_array = np.array(normalized_columns) # converting to numpy arrays for more efficient computation

loop_c = -1

for column in numeric_df_array.T:
    loop_c+=1
    x = column
    k2, p = stats.normaltest(x)
    alpha = 0.001
    print("p = {:g}".format(p))

    # rules for printing the normality output
    if p < alpha:
        test_result = "non_normal_distr"
        print("The null hypothesis can be rejected: non-normal distribution")

    else:
        test_result = "normal_distr"
        print("The null hypothesis cannot be rejected: normal distribution")
```

```
p = 5.41642e-23
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 1.64649e-201
The null hypothesis can be rejected: non-normal distribution
p = 0.00189662
The null hypothesis cannot be rejected: normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 3.97301e-248
The null hypothesis can be rejected: non-normal distribution
p = 4.44408e-248
The null hypothesis can be rejected: non-normal distribution
```

```
# select columns to not normalize
columns_to_notnormalize = numeric_df
columns_to_notnormalize.drop(columns_to_notnormalize.columns[col_for_normalization], axis=1, inplace=True)

# binding both the non-normalized and normalized columns
numeric_df_normalized = pd.concat([columns_to_notnormalize.reset_index(drop=True), normalized_columns], axis=1)

numeric_df_normalized.head()
```

Figure 7.18: Effective segmentation of workflows

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Workflow Documentation

When products and solutions are developed, they are mostly developed, monitored, deployed, and tested in a sandbox environment. To ensure a smooth deployment process in a new environment, we must provide sufficient support documents for technical, as well as non-technical, users. Workflow documentation includes requirements and design documents, product documentation, methodology documentation, installation guides, software user manuals, hardware and software requirements, troubleshooting management, and test documents. These are mostly required for a product or a solution development. We cannot just hand over a bunch of code to a client/user and ask them to get it running. Workflow documentation helps during the deployment and integration stages in a client/user environment, which is highly important for code reproducibility.

At a high level, data science project documentation can be divided into two segments:

- Product documentation
- Methodology documentation

Product documentation provides information on how each functionality is used in the UI/UX and the application of it. Product documentation can be further segmented into:

- Installation guides
- Software design and user manual
- Test documents
- Troubleshooting management

Methodology documentation provides information on the algorithms that are used, the methods, the solution approach, and so on.

Exercise 46: Missing Value Preprocessing with High Reproducibility

The aim of this exercise is to learn how to develop code for high reproducibility in terms of missing value treatment preprocessing. We will be using the UCI Bank and Marketing dataset taken from this link: https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson07/Dataset/bank/bank.csv.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Perform the following steps to find the missing value preprocessing reproducibility:

1. Import the required libraries and packages in the Jupyter notebook, as illustrated here:

```
import numpy as np
import pandas as pd
import collections
import random
```

2. Set the working directory of your choice as illustrated here:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

3. Import the dataset from **`back.csv`** to the Spark object using the **`read_csv`** function, as illustrated here:

```
df = pd.read_csv('bank.csv', sep=';')
```

4. Now, view the first five rows of the dataset using the head function:

```
df.head(5)
```

The output is as follows:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1	0	unknown	no
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339	4	failure	no
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330	1	failure	no
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1	0	unknown	no
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	may	226	1	-1	0	unknown	no

Figure 7.19: Bank dataset

As the dataset has no missing values, we have to introduce some into the dataset.

5. First, set the loop parameters, as illustrated here:

```
replaced = collections.defaultdict(set)
ix = [(row, col) for row in range(df.shape[0]) for col in range(df.
shape[1])]
random.shuffle(ix)
to_replace = int(round(.1*len(ix)))
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Create a **`for`** loop for generating missing values:

```
for row, col in ix:
    if len(replaced[row]) < df.shape[1] - 1:
        df.iloc[row, col] = np.nan
        to_replace -= 1
        replaced[row].add(col)
        if to_replace == 0:
            break
```

7. Use the following command to identify the missing values in the data by looking into each column's missing values:

```
print(df.isna().sum())
```

The output is as follows:

```
age          442
job          424
marital      460
education    466
default      441
balance      405
housing      467
loan         474
contact      454
day          455
month        461
duration     458
campaign     484
pdays        419
previous     428
poutcome     481
y            467
dtype: int64
```

Figure 7.20: Find the missing values

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

8. Define the range of **Interquartile Ranges** (**IQRs**) and apply them to the dataset to identify the outliers:

```
num = df._get_numeric_data()
Q1 = num.quantile(0.25)
Q3 = num.quantile(0.75)
IQR = Q3 - Q1
print(num < (Q1 - 1.5 * IQR))
print(num > (Q3 + 1.5 * IQR))
```

The output is as follows:

	age	balance	day	duration	campaign	pdays	previous
0	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False
7	False	False	False	False	False	False	False
8	False	False	False	False	False	False	False
9	False	False	False	False	False	False	False
10	False	False	False	False	False	False	False
11	False	False	False	False	False	False	False
12	False	False	False	False	False	False	False
13	False	False	False	False	False	False	False
14	False	False	False	False	False	False	False
15	False	False	False	False	False	False	False
16	False	False	False	False	False	False	False
17	False	False	False	False	False	False	False
18	False	False	False	False	False	False	False

Figure 7.21: Identifying outliers

Avoiding Repetition

We all know that the duplication or repetition of code is not a good practice. It becomes difficult to handle bugs, and the length of code increases. Different versions of the same code can lead to difficulty after a point, in terms of understanding which version is correct. For debugging, a change in one position needs to be reflected across the code. To avoid bad practices and write and maintain high-level code, let's learn about some best practices in the following sections.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Using Functions and Loops for Optimizing Code

A function confines a task which requires a set of steps that from a single of multiple inputs to single or multiple outputs and loops are used for repetitive tasks on the same block of code for a different set of sample or subsetted data. Functions can be written for a single variable, multiple variables, a DataFrame, or a multiple set of parameter inputs.

For example, let's say you need to carry out some kind of transformation for only numeric variables in a DataFrame or matrix. A function can be written for a single variable and it can be applied to all numeric columns, or it can be written for a DataFrame, where the function identifies the set of numeric variables and applies them to generate the output. Once a function is written, it can be applied any future similar application in the proceeding code. This reduces duplicate work.

The following are the challenges that need to be considered when writing a function:

- **Internal parameter changes**: There can be changes in the input parameter from one task to another. This is a common challenge. To handle this, you can mention the dynamic variables or objects in the function inputs when defining the inputs for a function.
- **Variations in the calculation process for a future task**: Write a function with internal functions that will not require many changes if any variations need to be captured. This way, rewriting the function for a new kind of task will be easy.
- **Avoiding loops in functions**: If a process needs to be carried out across many subsets of the data by row, functions can be directly applied in each loop. This way, your function will be not be constrained by repetitive blocks of code on the same data.
- **Handing changes in data type changes**: The return object in a function can be different for different tasks. Depending on the task, the return object can be converted to other data classes or data types as required. However, input data classes or data types can change from task to task. To handle this, you need to clearly provide comments to understand the inputs for a function.
- **Writing optimized functions**: Arrays are efficient when it comes to repetitive tasks such as loops or functions. In Python, using NumPy arrays generates very efficient data processing for most arithmetic operations.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Developing Libraries/Packages for Code/Algorithm Reuse

Packages or libraries encapsulate a collection of modules. They are highly dependable for code reproducibility and the modules that are generated. There are thousands of packages/libraries that are generated daily by developers and researchers around the globe. You can follow the package developing instructions from the Python project packaging guide for developing a new package (https://packaging.python.org/tutorials/packaging-projects/). This tutorial will provide you with information on how to upload and distribute your package publicly as well as for internal use.

Activity 14: Carry normalisation of data

The aim of this activity is to apply various preprocessing techniques that were learned in previous exercises and to develop a model using preprocessed data.

Now let's perform the following steps:

1. Import the required libraries and read the data from the `bank.csv` file.
2. Import the dataset and read the CSV file into the Spark object.

   Check the normality of the data–the following step is to identify the normality of data.
3. Segment the data numerically and categorically and perform distribution transformation on the numeric data.
4. Create a `for` loop that loops through each column to carry out a normality test to detect the normal distribution of data.
5. Create a power transformer. A power transformer will transform the data from a non-normal distribution to a normal distribution. The model developed will be used to transform the previously identified columns, which are non-normal.
6. Apply the created power transformer model to the non-normal data.
7. To develop a model, first split the data into training and testing for cross-validation, train the model, then predict the model in test data for cross-validation. Finally, generate a confusion matrix for cross-validation.

> **Note**
>
> The solution of this activity can be found on page 240.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

In this chapter, we have learned how to maintain code reproducibility from a data science perspective through structured standards and practices to avoid duplicate work using the Jupyter notebook.

We started by gaining an understanding of what reproducibility is and how it impacts research and data science work. We looked into areas where we can improve code reproducibility, particularly looking at how we can maintain effective coding standards in terms of data reproducibility. Following that, we looked at important coding standards and practices to avoid duplicate work using the effective management of code through the segmentation of workflows, by developing functions for all key tasks, and how we can generalize coding to create libraries and packages from a reusability standpoint.

In the next chapter, we will learn how to use all the functionalities we have learned about so far to generate a full analysis report. We will also learn how to use various PySpark functionalities for SQL operations and how to develop various visualization plots.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

8

Creating a Full Analysis Report

Learning Objectives

By the end of this chapter, you will be able to:

- Read data from different sources in Spark
- Perform SQL operations on a Spark DataFrame
- Generate statistical measurements in a consistent way
- Generate graphs and plots using Plotly
- Compile an analysis report incorporating all the previous steps and data

In this chapter, we will read the data using Spark, aggregating it, and extract the statistical measurements. We will also use the Pandas to generate graphs from aggregated data and form an analysis report.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Introduction

If you have been part of the data industry for a while, you will understand the challenge of working with different data sources, analyzing them, and presenting them in consumable business reports. When using Spark on Python, you may have to read data from various sources, such as flat files, REST APIs in JSON format, and so on.

In the real world, getting data in the right format is always a challenge and several SQL operations are required to gather data. Thus, it is mandatory for any data scientist to know how to handle different file formats and different sources, and to carry out basic SQL operations and present them in a consumable format.

This chapter provides common methods for reading different types of data, carrying out SQL operations on it, doing descriptive statistical analysis, and generating a full analysis report. We will start with understanding how to read different kinds of data into PySpark and will then generate various analyses and plots on it.

Reading Data in Spark from Different Data Sources

One of the advantages of Spark is the ability to read data from various data sources. However, this is not consistent and keeps changing with each Spark version. This section of the chapter will explain how to read files in CSV and JSON.

Exercise 47: Reading Data from a CSV File Using the PySpark Object

To read CSV data, you have to write the **`spark.read.csv("the file name with .csv")`** function. Here, we are reading the bank data that was used in the earlier chapters.

> **Note**
>
> The **`sep`** function is used here.

We have to ensure that the right **`sep`** function is used based on how the data is separated in the source data.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Now let's perform the following steps to read the data from the **bank.csv** file:

1. First, let's import the required packages into the Jupyter notebook:

```
import os
import pandas as pd
import numpy as np
import collections
from sklearn.base import TransformerMixin
import random
import pandas_profiling
```

2. Next, import all the required libraries, as illustrated:

```
import seaborn as sns
import time
import re
import os
import matplotlib.pyplot as plt
```

Now, use the **tick** themes, which will make our dataset more visible and provide higher contrast:

```
sns.set(style="ticks")
```

1. Now, change the working directory using the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

2. Let's import the libraries required for Spark to build the Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('ml-bank').getOrCreate()
```

3. Now, let's read the CSV data after creating the **df_csv** Spark object using the following command:

```
df_csv = spark.read.csv('bank.csv', sep=';', header = True, inferSchema =
True)
```

4. Print the schema using the following command:

```
df_csv.printSchema()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- day: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- y: string (nullable = true)
```

Figure 8.1: Bank schema

Reading JSON Data Using the PySpark Object

To read JSON data, you have to write the **`read.json("the file name with .json")`** function after setting the SQL context:

Read JSON file in PySpark

```
# read the json data file
df_json = sqlContext.read.json('bank.json')
```

Figure 8.2: Reading JSON file in PySpark

SQL Operations on a Spark DataFrame

A DataFrame in Spark is a distributed collection of rows and columns. It is the same as a table in a relational database or an Excel sheet. A Spark RDD/DataFrame is efficient at processing large amounts of data and has the ability to handle petabytes of data, whether structured or unstructured.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Spark optimizes queries on data by organizing the DataFrame into columns, which helps Spark understand the schema. Some of the most frequently used SQL operations include subsetting the data, merging the data, filtering, selecting specific columns, dropping columns, dropping all null values, and adding new columns, among others.

Exercise 48: Reading Data in PySpark and Carrying Out SQL Operations

For summary statistics of data, we can use the **`spark_df.describe().show()`** function, which will provide information on **`count`**, **`mean`**, **`standard deviation`**, **`max`**, and **`min`** for all the columns in the DataFrame.

For example, in the dataset that we have considered—the bank marketing dataset (https://raw.githubusercontent.com/TrainingByPackt/Big-Data-Analysis-with-Python/master/Lesson08/bank.csv)—the summary statistics data can be obtained as follows:

1. After creating a new Jupyter notebook, import all the required packages, as illustrated here:

```
import os
import pandas as pd
import numpy as np
```

2. Now, change the working directory using the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

3. Import all the libraries required for Spark to build the Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('ml-bank').getOrCreate()
```

4. Create and read the data from the CSV file using the Spark object, as illustrated:

```
spark_df = spark.read.csv('bank.csv', sep=';', header = True, inferSchema
= True)
```

5. Now let's print the first five rows from the Spark object using the following command:

```
spark_df.head(5)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
[Row(age=30, job='unemployed', marital='married', education='primary', default='no', balance=1787, housing='no', loan='no', con
tact='cellular', day=19, month='oct', duration=79, campaign=1, pdays=-1, previous=0, poutcome='unknown', y='no'),
 Row(age=33, job='services', marital='married', education='secondary', default='no', balance=4789, housing='yes', loan='yes', c
ontact='cellular', day=11, month='may', duration=220, campaign=1, pdays=339, previous=4, poutcome='failure', y='no'),
 Row(age=35, job='management', marital='single', education='tertiary', default='no', balance=1350, housing='yes', loan='no', co
ntact='cellular', day=16, month='apr', duration=185, campaign=1, pdays=330, previous=1, poutcome='failure', y='no'),
 Row(age=30, job='management', marital='married', education='tertiary', default='no', balance=1476, housing='yes', loan='yes',
contact='unknown', day=3, month='jun', duration=199, campaign=4, pdays=-1, previous=0, poutcome='unknown', y='no'),
 Row(age=59, job='blue-collar', marital='married', education='secondary', default='no', balance=0, housing='yes', loan='no', co
ntact='unknown', day=5, month='may', duration=226, campaign=1, pdays=-1, previous=0, poutcome='unknown', y='no')]
```

Figure 8.3: Bank data of first five rows (Unstructured)

6. The previous output is unstructured. Let's first identify the data types to proceed to get the structured data. Use the following command to print the datatype of each column:

```
spark_df.printSchema()
```

The output is as follows:

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- day: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- y: string (nullable = true)
```

Figure 8.4: Bank datatypes (Structured)

7. Now let's calculate the total number of rows and columns with names to get a clear idea of the data we have:

```
spark_df.count()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
4521
len(spark_df.columns), spark_df.columns
```

The output is as follows:

```
(17,
 ['age',
  'job',
  'marital',
  'education',
  'default',
  'balance',
  'housing',
  'loan',
  'contact',
  'day',
  'month',
  'duration',
  'campaign',
  'pdays',
  'previous',
  'poutcome',
  'y'])
```

Figure 8.5: Total number of rows and columns names

8. Print the summary statistics for the DataFrame using the following command:

```
spark_df.describe().show()
```

The output is as follows:

Summary statistics of numerical columns

```
# statistics for the dataframe
spark_df.describe().show()
```

```
+-------+------------------+-------+--------+---------+-------+------------------+-------+----+--------+
----+-----+------------------+------------------+------------------+------------------+--------+----+
|summary|               age|    job| marital|education|default|           balance|housing|loan| contact|
day|month|          duration|          campaign|             pdays|          previous|poutcome|   y|
+-------+------------------+-------+--------+---------+-------+------------------+-------+----+--------+
----+-----+------------------+------------------+------------------+------------------+--------+----+
|  count|              4521|   4521|    4521|     4521|   4521|              4521|   4521|4521|    4521|
4521| 4521|              4521|              4521|              4521|              4521|    4521|4521|
|   mean| 41.17009511170095|   null|    null|     null|   null|1422.6578190665782|   null|null|    null|
2842| null|263.96129174961294| 2.793629727936297|39.766644547666445|0.5425790754257908|    null|null|
| stddev|10.576210958711263|   null|    null|     null|   null|3009.6381424673395|   null|null|    null|
9934| null|259.85663262468216|3.1098066601885823|100.12112444301656|1.6935623506071211|    null|null|
|    min|                19| admin.|divorced|  primary|     no|             -3313|     no|  no|cellular|
1|  apr|                 4|                 1|                -1|                 0| failure|  no|
|    max|                87|unknown|  single|  unknown|    yes|             71188|    yes| yes| unknown|
31|  sep|              3025|                50|               871|                25| unknown| yes|
+-------+------------------+-------+--------+---------+-------+------------------+-------+----+--------+
----+-----+------------------+------------------+------------------+------------------+--------+----+
```

Figure 8.6: Summary statistics of numerical columns

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

To select multiple columns from a DataFrame, we can use the **spark_df.select('col1', 'col2', 'col3')** function. For example, let's select the first five rows from the **balance** and **y** columns using the following command:

```
spark_df.select('balance','y').show(5)
```

The output is as follows:

```
+-------+---+
|balance|  y|
+-------+---+
|   1787| no|
|   4789| no|
|   1350| no|
|   1476| no|
|      0| no|
+-------+---+
only showing top 5 rows
```

Figure 8.7: Data of the balance and y columns

9. To identify the relation between two variables in terms of their frequency of levels, **crosstab** can be used. To derive crosstab between two columns, we can use the **spark_df.crosstab('col1', col2)** function. Crosstab is carried out between two categorical variables and not between numerical variables:

```
spark_df.crosstab('y', 'marital').show()
```

```
+---------+--------+-------+------+
|y_marital|divorced|married|single|
+---------+--------+-------+------+
|      yes|      77|    277|   167|
|       no|     451|   2520|  1029|
+---------+--------+-------+------+
```

Figure 8.8: Pair wise frequency of categorical columns

10. Now, let's add a new column to the dataset:

```
# sample sets
sample1 = spark_df.sample(False, 0.2, 42)
sample2 = spark_df.sample(False, 0.2, 43)

# train set
train = spark_df.sample(False, 0.8, 44)
train.withColumn('balance_new', train.balance /2.0).
select('balance','balance_new').show(5)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
+-------+-----------+
|balance|balance_new|
+-------+-----------+
|   1787|      893.5|
|   4789|     2394.5|
|   1350|      675.0|
|   1476|      738.0|
|    747|      373.5|
+-------+-----------+
only showing top 5 rows
```

Figure 8.9: Data of newly added column

11. Drop the newly created column using the following command:

```
train.drop('balance_new)
```

Exercise 49: Creating and Merging Two DataFrames

In this exercise, we will extract and use the bank marketing data (https://archive.ics.uci.edu/ml/datasets/bank+marketing) from the UCI Machine Learning Repository. The objective is to carry out merge operations on a Spark DataFrame using PySpark.

The data is related to the direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact for the same client was required, in order to access whether the product (**bank term deposit**) would be (**yes**) or would not be (**no**) subscribed.

Now, let's create two DataFrames from the current bank marketing data and merge them on the basis of a primary key:

1. First, let's import the required header files in the Jupyter notebook:

```
import os
import pandas as pd
import numpy as np
import pyspark
```

2. Now, change the working directory using the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Import all the libraries required for Spark to build the Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('ml-bank').getOrCreate()
```

4. Read the data from the CSV files into a Spark object using the following command:

```
spark_df = spark.read.csv('bank.csv', sep=';', header = True, inferSchema
= True)
```

5. Print the first five rows from the Spark object:

```
spark_df.head(5)
```

The output is as follows:

```
[Row(age=30, job='unemployed', marital='married', education='primary', default='no', balance=1787, housing='no', loan='no', con
tact='cellular', day=19, month='oct', duration=79, campaign=1, pdays=-1, previous=0, poutcome='unknown', y='no'),
 Row(age=33, job='services', marital='married', education='secondary', default='no', balance=4789, housing='yes', loan='yes', c
ontact='cellular', day=11, month='may', duration=220, campaign=1, pdays=339, previous=4, poutcome='failure', y='no'),
 Row(age=35, job='management', marital='single', education='tertiary', default='no', balance=1350, housing='yes', loan='no', co
ntact='cellular', day=16, month='apr', duration=185, campaign=1, pdays=330, previous=1, poutcome='failure', y='no'),
 Row(age=30, job='management', marital='married', education='tertiary', default='no', balance=1476, housing='yes', loan='yes',
contact='unknown', day=3, month='jun', duration=199, campaign=4, pdays=-1, previous=0, poutcome='unknown', y='no'),
 Row(age=59, job='blue-collar', marital='married', education='secondary', default='no', balance=0, housing='yes', loan='no', co
ntact='unknown', day=5, month='may', duration=226, campaign=1, pdays=-1, previous=0, poutcome='unknown', y='no')]
```

Figure 8.10: Bank data of first five rows (Unstructured)

6. Now, to merge the two DataFrames using the primary key (ID), first, we will have to split it into two DataFrames.

7. First, add a new DataFrame with an **ID** column:

```
from pyspark.sql.functions import monotonically_increasing_id
train_with_id = spark_df.withColumn("ID", monotonically_increasing_id())
```

8. Then, create another column, **ID2**:

```
train_with_id = train_with_id.withColumn('ID2', train_with_id.ID)
```

9. Split the DataFrame using the following command:

```
train_with_id1 = train_with_id.drop('balance', "ID2")
train_with_id2 = train_with_id.select('balance', "ID2")
```

10. Now, change the ID column names of **train_with_id2**:

```
train_with_id2 = train_with_id2.withColumnRenamed("ID2", "ID")
```

11. Merge **train_with_id1** and **train_with_id2** using the following command:

```
train_merged = train_with_id1.join(train_with_id2, on=['ID'], how='left_
outer')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Exercise 50: Subsetting the DataFrame

In this exercise, we will extract and use the bank marketing data (https://archive.ics.uci.edu/ml/datasets/bank+marketing) from the UCI Machine Learning Repository. The objective is to carry out filter/subsetting operations on the Spark DataFrame using PySpark.

Let's subset the DataFrame where the balance is greater than `0` in the bank marketing data:

1. First, let's import the required header files in the Jupyter notebook:

```
import os
import pandas as pd
import numpy as np
import pyspark
```

2. Now, change the working directory using the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

3. Import all the libraries required for Spark to build the Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('ml-bank').getOrCreate()
```

4. Now, read the CSV data as a Spark object using the following command:

```
spark_df = spark.read.csv('bank.csv', sep=';', header = True, inferSchema
= True)
```

5. Let's run SQL queries to subset and filter the DataFrame:

```
train_subsetted = spark_df.filter(spark_df.balance > 0.0)
pd.DataFrame(train_subsetted.head(5))
```

The output is as follows:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1	0	unknown	no
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339	4	failure	no
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330	1	failure	no
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1	0	unknown	no
4	35	management	single	tertiary	no	747	no	no	cellular	23	feb	141	2	176	3	failure	no

Figure 8.11: Filtered DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Generating Statistical Measurements

Python is a general-purpose language with statistical modules. A lot of statistical analysis, such as carrying out descriptive analysis, which includes identifying the distribution of data for numeric variables, generating a correlation matrix, the frequency of levels in categorical variables with identifying mode and so on, can be carried out in Python. The following is an example of correlation:

Segment Numeric Data and Generate Correlation Matrix for Numeric Variables

```
# select numeric columns
df = pandas_df._get_numeric_data()

# call required packages
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="ticks")


# develop a correlation matrix and represent it using heatmap
corr_df = df.corr()
sns.heatmap(corr_df, xticklabels=corr_df.columns.values, yticklabels=corr_df.columns.values, annot = True, annot_kws={'s
heat_map=plt.gcf(); heat_map.set_size_inches(10,5)
plt.xticks(fontsize=10); plt.yticks(fontsize=10); plt.show()
```

```
<Figure size 1000x500 with 2 Axes>
```

Generate Correlation Matrix Output

```
# print the correlations
print(corr_df)
```

```
               age   balance       day  duration  campaign     pdays  previous
age       1.000000  0.083820 -0.017853 -0.002367 -0.005148 -0.008894 -0.003511
balance   0.083820  1.000000 -0.008677 -0.015950 -0.009976  0.009437  0.026196
day      -0.017853 -0.008677  1.000000 -0.024629  0.160706 -0.094352 -0.059114
duration -0.002367 -0.015950 -0.024629  1.000000 -0.068382  0.010380  0.018080
campaign -0.005148 -0.009976  0.160706 -0.068382  1.000000 -0.093137 -0.067833
pdays    -0.008894  0.009437 -0.094352  0.010380 -0.093137  1.000000  0.577562
previous -0.003511  0.026196 -0.059114  0.018080 -0.067833  0.577562  1.000000
```

Figure 8.12: Segment numeric data and correlation matrix output

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Identifying the distribution of data and normalizing it is important for parametric models such as **linear regression** and **support vector machines**. These algorithms assume the data to be normally distributed. If data is not normally distributed, it can lead to bias in the data. In the following example, we will identify the distribution of data through a normality test and then apply a transformation using the **yeo-johnson** method to normalize the data:

Identify the Distribution of Data - Normality Test

```
# loop for identifying the columns with non-normal distribution
numeric_df_array = np.array(df) # converting to numpy arrays for more efficient computation

loop_c = -1
col_for_normalization = list()

for column in numeric_df_array.T:
    loop_c+=1
    x = column
    k2, p = stats.normaltest(x)
    alpha = 0.001
    print("p = {:g}".format(p))

    # rules for printing the normality output
    if p < alpha:
        test_result = "non_normal_distr"
        col_for_normalization.append((loop_c)) # applicable if yeo-johnson is used

        #if min(x) > 0: # applicable if box-cox is used
            #col_for_normalization.append((loop_c)) # applicable if box-cox is used
        print("The null hypothesis can be rejected: non-normal distribution")

    else:
        test_result = "normal_distr"
        print("The null hypothesis cannot be rejected: normal distribution")
```

Figure 8.13: Identifying the distribution of the data—Normality test

The identified variables are then normalized using **yeo-johnson** or the **box-cox** method.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Generating the importance of features is important in a data science project where predictive techniques are used. This broadly falls under statistical analysis as various statistical techniques are used to identify the important variables. One of the methods that's used here is **Boruta**, which is a wrap-around **RandomForest** algorithm for variable importance analysis. For this, we will be using the **BorutaPy** package:

Feature Importance

```
# df = pd.read_csv('Daily_Demand_Forecasting_Orders.csv', sep=';')
df = pd.read_csv('bank.csv', sep=';')
```

```
# import DecisionTreeClassifier from sklearn and BorutaPy from boruta
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from boruta import BorutaPy

# transform all categorical data types to integers (hot-encoding)
for col_name in df.columns:
    if(df[col_name].dtype == 'object'):
        df[col_name]= df[col_name].astype('category')
        df[col_name] = df[col_name].cat.codes

# generate separate dataframes for IVs and DV (target variable)
X = df.drop(['y'], axis=1).values
Y = df['y'].values
```

```
# build RandomForestClassifier, Boruta models and related parameter
rfc = RandomForestClassifier(n_estimators=10, n_jobs=4, class_weight='balanced', max_depth=3)
boruta_selector = BorutaPy(rfc, n_estimators='auto', verbose=2)

# fit Boruta algorithm
boruta_selector.fit(X, Y)
```

Figure 8.14: Feature importance

Activity 15: Generating Visualization Using Plotly

In this activity, we will extract and use the bank marketing data from the UCI Machine Learning Repository. The objective is to generate visualizations using Plotly in Python.

> **Note**
>
> Plotly's Python graphing library makes interactive, publication-quality graphs.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Perform the following steps to generate the visualization using Plotly:

1. Import the required libraries and packages into the Jupyter notebook.
2. Import the libraries required for Plotly to visualize the data visualization:

```
import plotly.graph_objs as go
from plotly.plotly import iplot
import plotly as py
```

3. Read the data from the **bank.csv** file into the Spark DataFrame.
4. Check the Plotly version that you are running on your system. Make sure you are running the updated version. Use the **pip install plotly --upgrade** command and then run the following code:

```
from plotly import __version__
from plotly.offline import download_plotlyjs, init_notebook_mode, plot,
iplot

print(__version__) # requires version >= 1.9.0
```

The output is as follows:

```
3.7.1
```

5. Now import the required libraries to plot the graphs using Plotly:

```
import plotly.plotly as py
import plotly.graph_objs as go
from plotly.plotly import iplot
init_notebook_mode(connected=True)
```

6. Set the Plotly credentials in the following command, as illustrated here:

```
plotly.tools.set_credentials_file(username='Your_Username', api_key='Your_
API_Key')
```

> **Note**
>
> To generate an API key for Plotly, sign up for an account and navigate to https://plot.ly/settings#/. Click on the **API Keys** option and then click on the **Regenerate Key** option.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Now, plot each of the following graphs using Plotly:

   Bar graph:

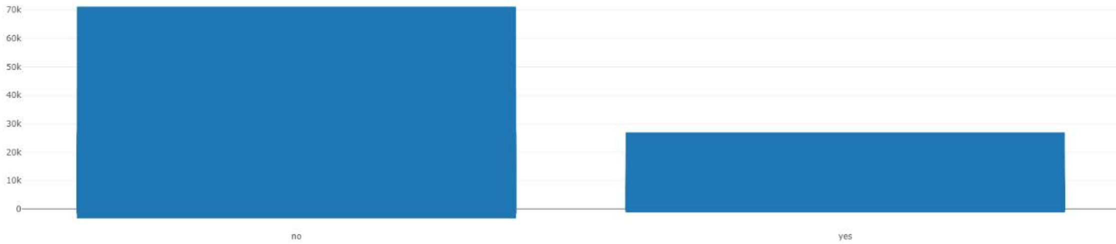


Figure 8.15: Bar graph of bank data

Scatter plot:

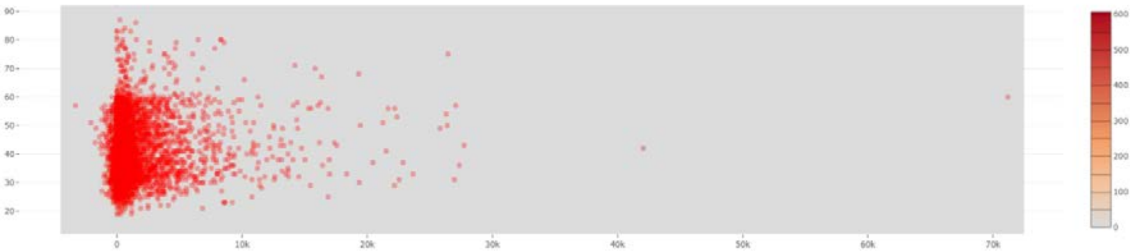


Figure 8.16: Scatter plot of bank data

Boxplot:

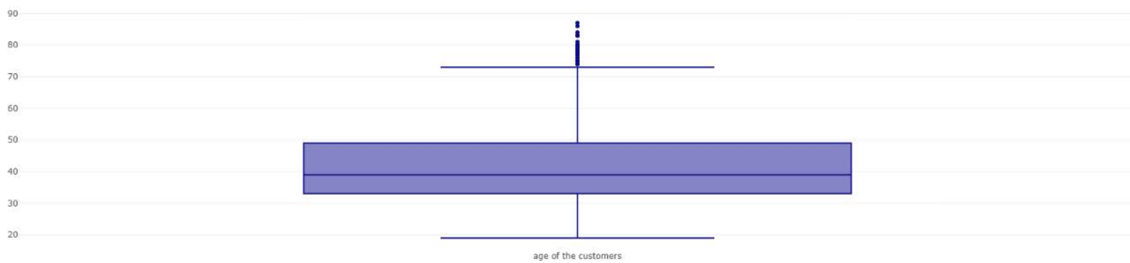


Figure 8.17: Boxplot of bank data

> **Note**
>
> The solution for this activity can be found on page 248.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Summary

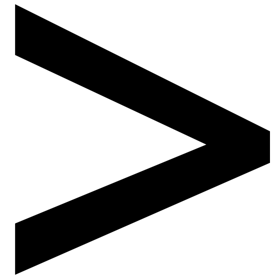
In this chapter, we learned how to import data from various sources into a Spark environment as a Spark DataFrame. In addition, we learned how to carry out various SQL operations on that DataFrame, and how to generate various statistical measures, such as correlation analysis, identifying the distribution of data, building a feature importance model, and so on. We also looked into how to generate effective graphs using Plotly offline, where you can generate various plots to develop an analysis report.

This book has hopefully offered a stimulating journey through big data. We started with Python and covered several libraries that are part of the Python data science stack: NumPy and Pandas, We also looked at home we can use Jupyter notebooks. We then saw how to create informative data visualizations, with some guiding principles on what is a good graph, and used Matplotlib and Seaborn to materialize the figures. Then we made a start with the Big Data tools - Hadoop and Spark, thereby understanding the principles and the basic operations.

We have seen how we can use DataFrames in Spark to manipulate data, and have about utilize concepts such as correlation and dimensionality reduction to better understand our data. The book has also covered reproducibility, so that the analysis created can be supported and better replicated when needed, and we finished our journey with a final report. We hope that the subjects covered, and the practical examples in this book will help you in all areas of your own data journey.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use



Appendix

About

This section is included to assist the students in performing the activities in the book. It includes detailed steps that are to be performed by the students to achieve the objectives of the activities.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Chapter 01: The Python Data Science Stack

Activity 1: IPython and Jupyter

1. Open the **python_script_student.py** file in a text editor, copy the contents to a notebook in IPython, and execute the operations.
2. Copy and paste the code from the Python script into a Jupyter notebook:

```
import numpy as np

def square_plus(x, c):
    return np.power(x, 2) + c
```

3. Now, update the values of the **x** and **c** variables. Then, change the definition of the function:

```
x = 10
c = 100

result = square_plus(x, c)
print(result)
```

   The output is as follows:

```
200
```

Activity 2: Working with Data Problems

1. Import pandas and NumPy library:

```
import pandas as pd
import numpy as np
```

2. Read the RadNet dataset from the U.S. Environmental Protection Agency, available from the Socrata project:

```
url = "https://opendata.socrata.com/api/views/cf4r-dfwe/rows.
csv?accessType=DOWNLOAD"
df = pd.read_csv(url)
```

3. Create a list with numeric columns for radionuclides in the RadNet dataset:

```
columns = df.columns
id_cols = ['State', 'Location', "Date Posted", 'Date Collected', 'Sample
Type', 'Unit']
columns = list(set(columns) - set(id_cols))
columns
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

4. Use the **apply** method on one column, with a **lambda** function that compares the **Non-detect** string:

```
df['Cs-134'] = df['Cs-134'].apply(lambda x: np.nan if x == "Non-detect"
else x)
df.head()
```

The output is as follows:

	State	Location	Date Posted	Date Collected	Sample Type	Unit	Ba-140	Co-60	Cs-134
0	ID	Boise	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
1	ID	Boise	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
2	AK	Juneau	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	0.0057
3	AK	Nome	03/30/2011	03/22/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN
4	AK	Nome	03/30/2011	03/23/2011	Air Filter	pCi/m3	Non-detect	Non-detect	NaN

Figure 1.19: DataFrame after applying the lambda function

5. Replace the text values with **NaN** in one column with **np.nan**:

```
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'Non-detect' else x)
df.loc[:, columns] = df.loc[:, columns].applymap(lambda x: np.nan if x ==
'ND' else x)
```

6. Use the same lambda comparison and use the **applymap** method on several columns at the same time, using the list created in the first step:

```
df.loc[:, ['State', 'Location', 'Sample Type', 'Unit']] = df.loc[:,
['State', 'Location', 'Sample Type', 'Unit']].applymap(lambda x:
x.strip())
```

7. Create a list of the remaining columns that are not numeric:

```
df.dtypes
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
State              object
Location           object
Date Posted        object
Date Collected     object
Sample Type        object
Unit               object
Ba-140             object
Co-60              object
Cs-134             object
Cs-136             object
Cs-137             object
I-131              object
I-132              object
I-133              object
Te-129             object
Te-129m            object
Te-132             object
dtype: object
```

Figure 1.20: List of columns and their type

8. Convert the DataFrame objects into floats using the **to_numeric** function:

```
df['Date Posted'] = pd.to_datetime(df['Date Posted'])
df['Date Collected'] = pd.to_datetime(df['Date Collected'])
for col in columns:
    df[col] = pd.to_numeric(df[col])
df.dtypes
```

The output is as follows:

```
State                       object
Location                    object
Date Posted         datetime64[ns]
Date Collected      datetime64[ns]
Sample Type                 object
Unit                        object
Ba-140                     float64
Co-60                      float64
Cs-134                     float64
Cs-136                     float64
Cs-137                     float64
I-131                      float64
I-132                      float64
I-133                      float64
Te-129                     float64
Te-129m                    float64
Te-132                     float64
dtype: object
```

Figure 1.21: List of columns and their type

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Using the selection and filtering methods, verify that the names of the string columns don't have any spaces:

```
df['Date Posted'] = pd.to_datetime(df['Date Posted'])
df['Date Collected'] = pd.to_datetime(df['Date Collected'])
for col in columns:
    df[col] = pd.to_numeric(df[col])
df.dtypes
```

The output is as follows:

```
State                     category
Location                  category
Date Posted         datetime64[ns]
Date Collected      datetime64[ns]
Sample Type               category
Unit                      category
```

Figure 1.22: DataFrame after applying the selection and filtering method

Activity 3: Plotting Data with Pandas

1. Use the RadNet DataFrame that we have been working with.
2. Fix all the data type problems, as we saw before.
3. Create a plot with a filter per **`Location`**, selecting the city of **`San Bernardino`**, and one radionuclide, with the x-axis set to the **`date`** and the y-axis with radionuclide **`I-131`**:

```
df.loc[df.Location == 'San Bernardino'].plot(x='Date Collected', y='I-
131')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

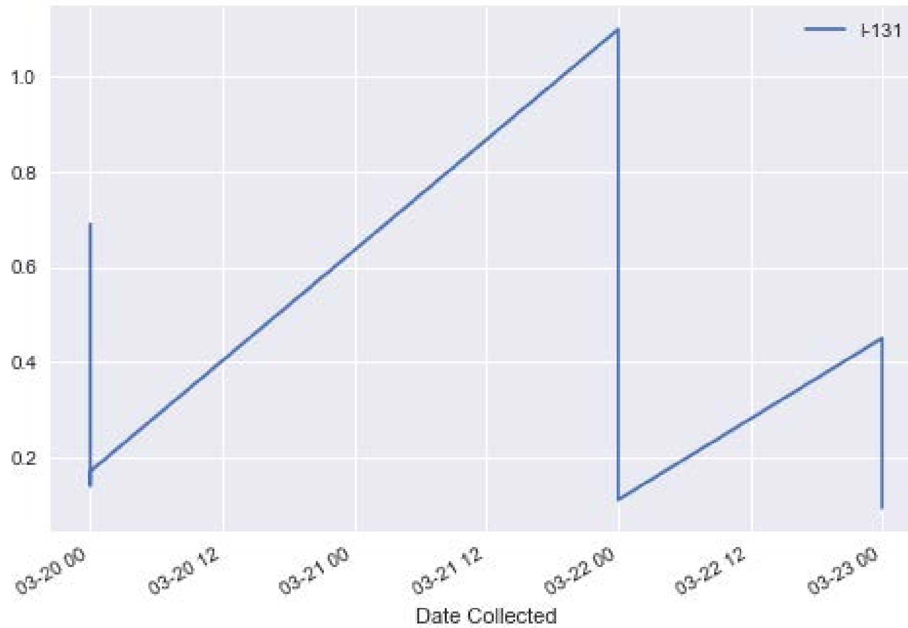


Figure 1.23: Plot of Date collected vs I-131

4. Create a scatter plot with the concentration of two related radionuclides, **I-131** and **I-132**:

```
fig, ax = plt.subplots()
ax.scatter(x=df['I-131'], y=df['I-132'])
_ = ax.set(
    xlabel='I-131',
    ylabel='I-132',
    title='Comparison between concentrations of I-131 and I-132'
)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

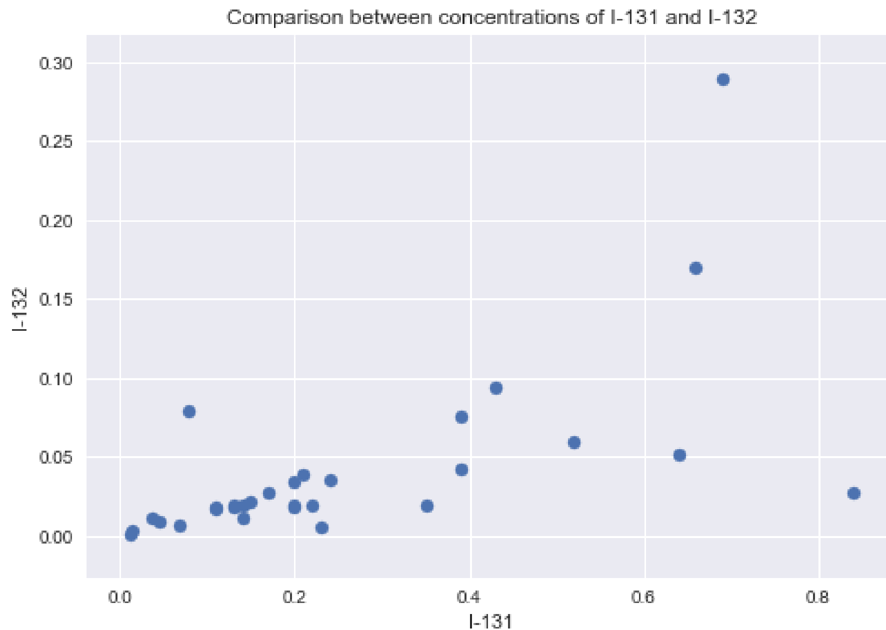


Figure 1.24: Plot of concentration of I-131 and I-132

Chapter 02: Statistical Visualizations Using Matplotlib and Seaborn

Activity 4: Line Graphs with the Object-Oriented API and Pandas DataFrames

1. Import the required libraries in the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data"
df = pd.read_csv(url)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Provide the column names to simplify the dataset, as illustrated here:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
df.head()
```

The plot is as follows:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Figure 2.29: The auto-mpg DataFrame

4. Convert the **`horsepower`** and **`year`** data types to float and integer using the following command:

```
df.loc[df.horsepower == '?', 'horsepower'] = np.nan
df['horsepower'] = pd.to_numeric(df['horsepower'])
df['full_date'] = pd.to_datetime(df.year, format='%y')
df['year'] = df['full_date'].dt.year
```

5. Let's display the data types:

```
df.dtypes
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
mpg                    float64
cylinders                int64
displacement           float64
horsepower             float64
weight                 float64
acceleration           float64
year                     int64
origin                   int64
name                    object
full_date       datetime64[ns]
dtype: object
```

Figure 2.30: The data types

6. Now plot the average **horsepower** per **year** using the following command:

```
df.groupby('year')['horsepower'].mean().plot()
```

The output is as follows:

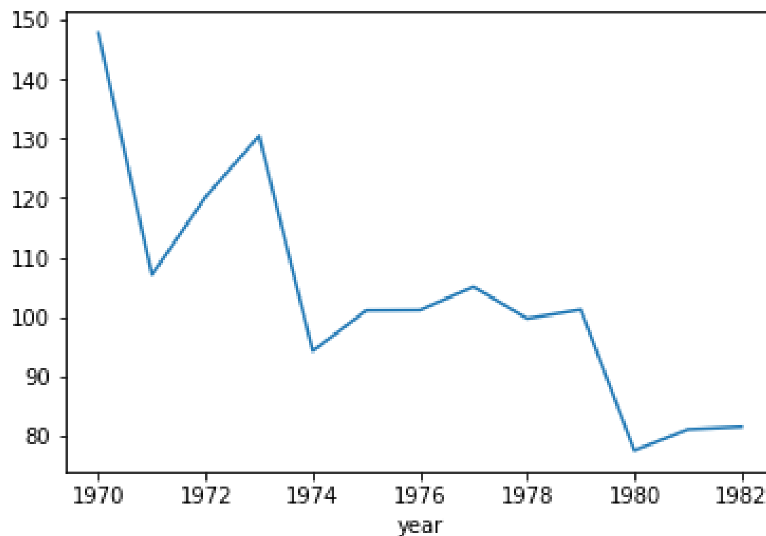


Figure 2.31: Line plot

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 5: Understanding Relationships of Variables Using Scatter Plots

1. Import the required libraries into the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data"
df = pd.read_csv(url)
```

2. Provide the column names to simplify the dataset as illustrated here:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
df.head()
```

The plot is as follows:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Figure 2.32: Auto-mpg DataFrame

4. Now plot the scatter plot using the **`scatter`** method:

```
fig, ax = plt.subplots()
ax.scatter(x = df['horsepower'], y=df['weight'])
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output will be as follows:

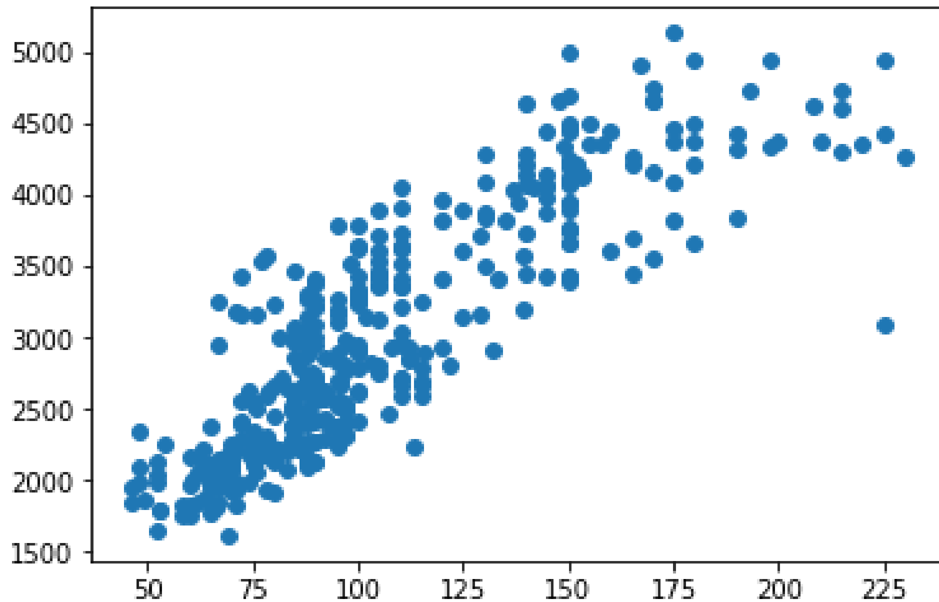


Figure 2.33: Scatter plot using the scatter method

Activity 6: Exporting a Graph to a File on Disk

1. Import the required libraries in the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data"
df = pd.read_csv(url)
```

2. Provide the column names to simplify the dataset, as illustrated here:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
```

4. Create a bar plot using the following command:

```
fig, ax = plt.subplots()
df.weight.plot(kind='hist', ax=ax)
```

The output is as follows:

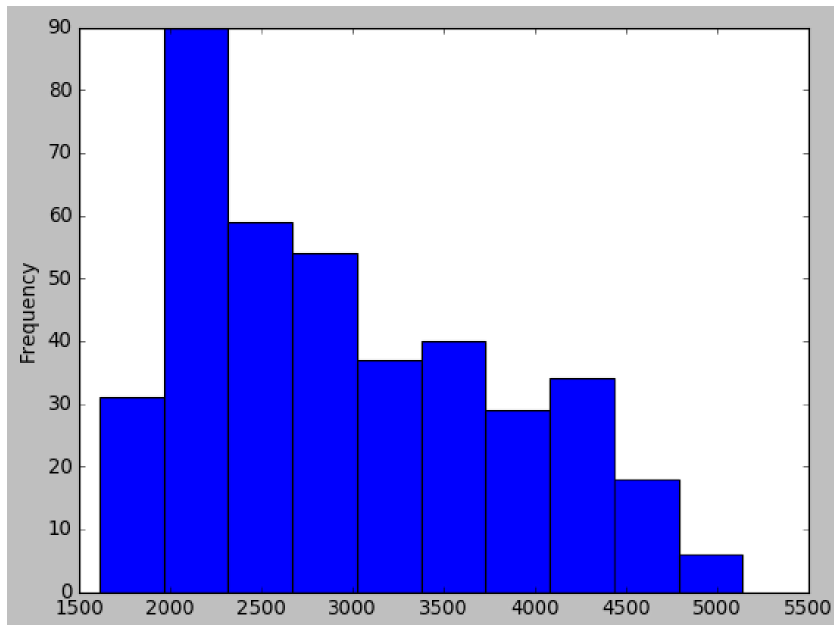


Figure 2.34: Bar plot

5. Export it to a PNG file using the **`savefig`** function:

```
fig.savefig('weight_hist.png')
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Activity 7: Complete Plot Design

1. Import the required libraries in the Jupyter notebook and read the dataset from the Auto-MPG dataset repository:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/
auto-mpg.data"
df = pd.read_csv(url)
```

2. Provide the column names to simplify the dataset, as illustrated here:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
'weight', 'acceleration', 'year', 'origin', 'name']
```

3. Now read the new dataset with column names and display it:

```
df = pd.read_csv(url, names= column_names, delim_whitespace=True)
```

4. Perform GroupBy on **`year`** and **`cylinders`**, and unset the option to use them as indexes:

```
df_g = df.groupby(['year', 'cylinders'], as_index=False)
```

5. Calculate the average miles per gallon over the grouping and set **`year`** as index:

```
df_g = df_g.mpg.mean()
```

6. Set year as the DataFrame index:

```
df_g = df_g.set_index(df_g.year)
```

7. Create the figure and axes using the object-oriented API:

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots()
```

8. Group the **`df_g`** dataset by **`cylinders`** and plot the miles per gallon variable using the axes created with size (**10**,**8**):

```
df = df.convert_objects(convert_numeric=True)
df_g = df.groupby(['year', 'cylinders'], as_index=False).horsepower.mean()
df_g = df_g.set_index(df_g.year)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Set the **title**, **x** label, and **y** label on the axes:

```
fig, axes = plt.subplots()
df_g.groupby('cylinders').horsepower.plot(axes=axes, figsize=(12,10))
_ = axes.set(
    title="Average car power per year",
    xlabel="Year",
    ylabel="Power (horsepower)"

)
```

The output is as follows:

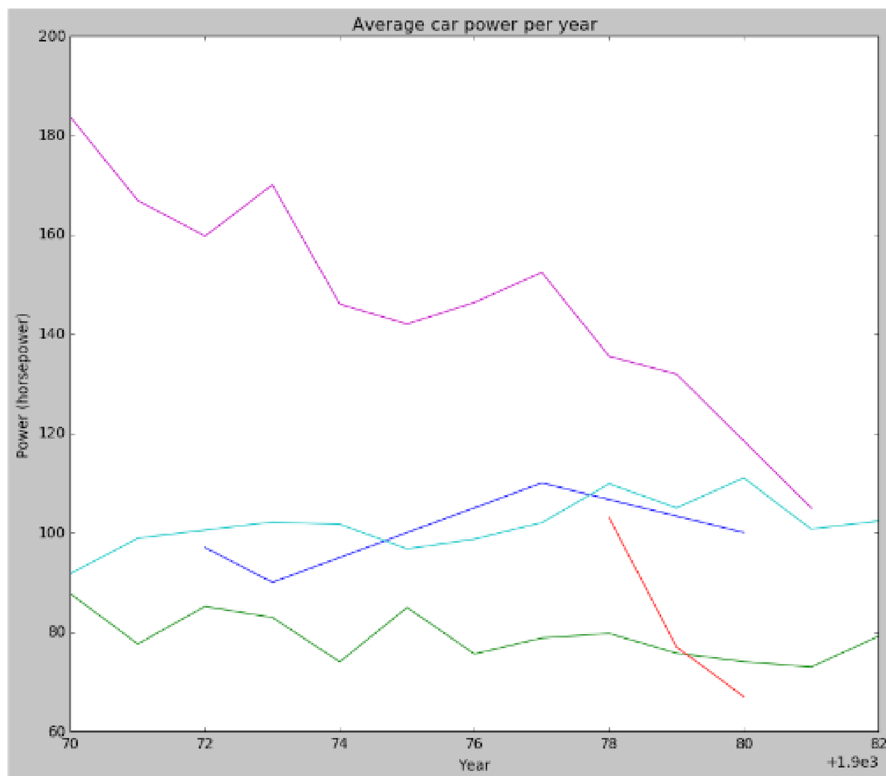


Figure 2.35: Line plot for average car power per year (without legends)

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

10. Include legends, as follows:

```
axes.legend(title='Cylinders', fancybox=True)
```

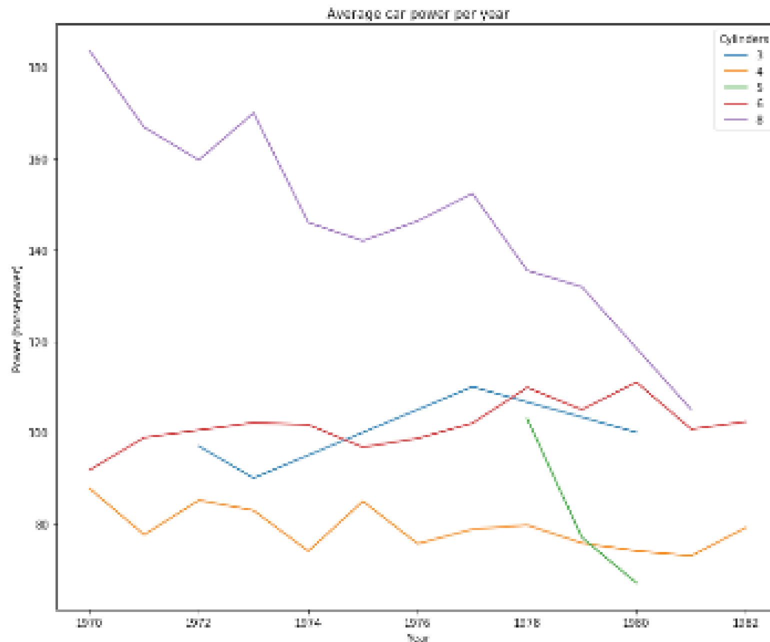


Figure 2.36: Line plot for average car power per year (with legends)

11. Save the figure to disk as a PNG file:

```
fig.savefig('mpg_cylinder_year.png')
```

Chapter 03: Working with Big Data Frameworks

Activity 8: Parsing Text

1. Read the text files into the Spark object using the **`text`** method:

```
rdd_df = spark.read.text("/localdata/myfile.txt").rdd
```

To parse the file that we are reading, we will use lambda functions and Spark operations such as **`map`**, **`flatMap`**, and **`reduceByKey`**. **`flatmap`** applies a function to all elements of an RDD, flattens the results, and returns the transformed RDD. **`reduceByKey`** merges the values based on the given key, combining the values. With these functions, we can count the number of lines and words in the text.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Extract the **lines** from the text using the following command:

```
lines = rdd_df.map(lambda line: line[0])
```

3. This splits each line in the file as an entry in the list. To check the result, you can use the **collect** method, which gathers all data back to the driver process:

```
lines.collect()
```

4. Now, let's count the number of lines, using the **count** method:

```
lines.count()
```

> **Note**
>
> Be careful when using the **collect** method! If the DataFrame or RDD being collected is larger than the memory of the local driver, Spark will throw an error.

5. Now, let's first split each line into words, breaking it by the space around it, and combining all elements, removing words in uppercase:

```
splits = lines.flatMap(lambda x: x.split(' '))
lower_splits = splits.map(lambda x: x.lower())
```

6. Let's also remove the *stop words*. We could use a more consistent stop words list from **NLTK**, but for now, we will row our own:

```
stop_words = ['of', 'a', 'and', 'to']
```

7. Use the following command to remove the stop words from our token list:

```
tokens = lower_splits.filter(lambda x: x and x not in stop_words)
```

We can now process our token list and count the unique words. The idea is to generate a list of tuples, where the first element is the **token** and the second element is the **count** of that particular token.

8. First, let's **map** our token to a list:

```
token_list = tokens.map(lambda x: [x, 1])
```

9. Use the **reduceByKey** operation, which will apply the operation to each of the lists:

```
count = token_list.reduceByKey(add).sortBy(lambda x: x[1],
ascending=False)
count.collect()
```

Remember, collect all data back to the driver node! Always check whether there is enough memory by using tools such as **top** and **htop**.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Chapter 04: Diving Deeper with Spark

Activity 9: Getting Started with Spark DataFrames

If you are using Google Collab to run the Jupyter notebook, add these lines to ensure you have set the environment:

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.0/spark-2.4.0-bin-hadoop2.7.tgz
!tar xf spark-2.4.0-bin-hadoop2.7.tgz
!pip install -q findspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.2-bin-hadoop2.7"
```

Install findspark if not installed using the following command:

```
pip install -q findspark
```

1. To create a sample DataFrame by manually specifying the schema, importing findspark module to connect Jupyter with Spark:

```
import findspark
findspark.init()
import pyspark
import os
```

2. Create the **SparkContext** and **SQLContext** using the following command:

```
sc = pyspark.SparkContext()
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)

from pyspark.sql import *
na_schema = Row("Name","Subject","Marks")
row1 = na_schema("Ankit", "Science",95)
row2 = na_schema("Ankit", "Maths", 86)
row3 = na_schema("Preity", "Maths", 92)
na_list = [row1, row2, row3]
df_na = sqlc.createDataFrame(na_list)
type(df_na)
```

The output is as follows:

```
pyspark.sql.dataframe.DataFrame
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Check the DataFrame using the following command:

```
df_na.show()
```

The output is as follows:

```
+------+-------+-----+
|  Name|Subject|Marks|
+------+-------+-----+
| Ankit|Science|   95|
| Ankit|  Maths|   86|
|Preity|  Maths|   92|
+------+-------+-----+
```

Figure 4.29: Sample DataFrame

4. Create a sample DataFrame from an existing RDD. First creating RDD as illustrated here:

```
data = [("Ankit","Science",95),("Preity","Maths",86),("Ankit","Maths",86)]
data_rdd = sc.parallelize(data)
type(data_rdd)
```

The output is as follows:

```
pyspark.rdd.RDD
```

5. Converting RDD to DataFrame using the following command:

```
data_df = sqlc.createDataFrame(data_rdd)
data_df.show()
```

The output is as follows:

```
+------+-------+---+
|    _1|     _2| _3|
+------+-------+---+
| Ankit|Science| 95|
|Preity|  Maths| 86|
| Ankit|  Maths| 86|
+------+-------+---+
```

Figure 4.30: RDD to DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Create a sample DataFrame by reading the data from a CSV file:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header='true',
inferschema='true').load('mtcars.csv')
type(df)
```

The output is as follows:

```
pyspark.sql.dataframe.DataFrame
```

7. Print first seven rows of the DataFrame:

```
df.show(7)
```

The output is as follows:

```
+----+---+-----+---+----+-----+-----+---+---+----+----+
| mpg|cyl| disp| hp|drat|   wt| qsec| vs| am|gear|carb|
+----+---+-----+---+----+-----+-----+---+---+----+----+
|21.0|  6|160.0|110| 3.9| 2.62|16.46|  0|  1|   4|   4|
|21.0|  6|160.0|110| 3.9|2.875|17.02|  0|  1|   4|   4|
|22.8|  4|108.0| 93|3.85| 2.32|18.61|  1|  1|   4|   1|
|21.4|  6|258.0|110|3.08|3.215|19.44|  1|  0|   3|   1|
|18.7|  8|360.0|175|3.15| 3.44|17.02|  0|  0|   3|   2|
|18.1|  6|225.0|105|2.76| 3.46|20.22|  1|  0|   3|   1|
|14.3|  8|360.0|245|3.21| 3.57|15.84|  0|  0|   3|   4|
+----+---+-----+---+----+-----+-----+---+---+----+----+
only showing top 7 rows
```

Figure 4.31: First seven rows of the DataFrame

8. Print the schema of the DataFrame:

```
df.printSchema()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. The output is as follows:

```
root
 |-- mpg: double (nullable = true)
 |-- cyl: integer (nullable = true)
 |-- disp: double (nullable = true)
 |-- hp: integer (nullable = true)
 |-- drat: double (nullable = true)
 |-- wt: double (nullable = true)
 |-- qsec: double (nullable = true)
 |-- vs: integer (nullable = true)
 |-- am: integer (nullable = true)
 |-- gear: integer (nullable = true)
 |-- carb: integer (nullable = true)
```

Figure 4.32: Schema of the DataFrame

10. Print the number of columns and rows in DataFrame:

```
print('number of rows:'+ str(df.count()))
print('number of columns:'+ str(len(df.columns)))
```

The output is as follows:

```
number of rows:32
number of columns:11
```

11. Print the summary statistics of DataFrame and any two individual columns:

```
df.describe().show()
```

The output is as follows:

summary	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
count	32	32	32	32	32	32	32	32	32	32	32
mean	20.090624999999996	6.1875	230.72187500000004	146.6875	3.5965625000000006	3.2172499999999995	17.848750000000003	0.4375	0.40625	3.6875	2.8125
stddev	6.026948052089103	1.7859216469465444	123.93869383138195	68.56286848932059	0.5346787360709716	0.9784574429896968	1.7869432360968436	0.5040161287741853	0.49899091723584604	0.7378040652569471	1.6151999776318522
min	10.4	4	71.1	52	2.76	1.513	14.5	0	0	3	1
max	33.9	8	472.0	335	4.93	5.424	22.9	1	1	5	8

Figure 4.33: Summary statistics of DataFrame

Print the summary of any two columns:

```
df.describe(['mpg','cyl']).show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
+-------+-----------------+------------------+
|summary|              mpg|               cyl|
+-------+-----------------+------------------+
|  count|               32|                32|
|   mean|20.090624999999996|            6.1875|
| stddev| 6.026948052089103|1.7859216469465444|
|    min|             10.4|                 4|
|    max|             33.9|                 8|
+-------+-----------------+------------------+
```

Figure 4.34: Summary statistics of mpg and cyl columns

12. Write first seen rows of the sample DataFrame in a CSV file:

```
df_p = df.toPandas()
df_p.head(7).to_csv("mtcars_head.csv")
```

Activity 10: Data Manipulation with Spark DataFrames

1. Install the packages as illustrated here:

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://www-us.apache.org/dist/spark/spark-2.4.0/spark-2.4.0-bin-hadoop2.7.tgz
!tar xf spark-2.4.0-bin-hadoop2.7.tgz
!pip install -q findspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.0-bin-hadoop2.7"
```

2. Then, import the **findspark** module to connect the Jupyter with Spark use the following command:

```
import findspark
findspark.init()
import pyspark
import os
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Now, create the **SparkContext** and **SQLContext** as illustrated here:

```
sc = pyspark.SparkContext()
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

4. Create a DataFrame in Spark as illustrated here:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header='true',
inferschema='true').load('mtcars.csv')
df.show(4)
```

The output is as follows:

```
+--------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|         model| mpg|cyl| disp| hp|drat|   wt| qsec| vs| am|gear|carb|
+--------------+----+---+-----+---+----+-----+-----+---+---+----+----+
|     Mazda RX4|21.0|  6|160.0|110| 3.9| 2.62|16.46|  0|  1|   4|   4|
| Mazda RX4 Wag|21.0|  6|160.0|110| 3.9|2.875|17.02|  0|  1|   4|   4|
|    Datsun 710|22.8|  4|108.0| 93|3.85| 2.32|18.61|  1|  1|   4|   1|
|Hornet 4 Drive|21.4|  6|258.0|110|3.08|3.215|19.44|  1|  0|   3|   1|
+--------------+----+---+-----+---+----+-----+-----+---+---+----+----+
only showing top 4 rows
```

Figure 4.35: DataFrame in Spark

5. Rename any five columns of DataFrame using the following command:

```
data = df
new_names = ['mpg_new', 'cyl_new', 'disp_new', 'hp_new', 'drat_new']
for i,z in zip(data.columns[0:5],new_names):
    data = data.withColumnRenamed(str(i),str(z))

data.columns
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
['mpg_new',
 'cyl_new',
 'disp_new',
 'hp_new',
 'drat_new',
 'drat',
 'wt',
 'qsec',
 'vs',
 'am',
 'gear',
 'carb']
```

Figure 4.36: Columns of DataFrame

6. Select any two numeric and one categorical column from the DataFrame:

```
data = df.select(['cyl','mpg','hp'])
data.show(5)
```

The output is as follows:

```
+---+----+---+
|cyl| mpg| hp|
+---+----+---+
|  6|21.0|110|
|  6|21.0|110|
|  4|22.8| 93|
|  6|21.4|110|
|  8|18.7|175|
+---+----+---+
only showing top 5 rows
```

Figure 4.37: Two numeric and one categorical column from the DataFrame

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

7. Count the number of distinct categories in the categorical variable:

```
data.select('cyl').distinct().count() #3
```

8. Create two new columns in DataFrame by summing up and multiplying together the two numerical columns:

```
data = data.withColumn('colsum',(df['mpg'] + df['hp']))
data = data.withColumn('colproduct',(df['mpg'] * df['hp']))
data.show(5)
```

The output is as follows:

```
+---+------+----------+
|cyl|colsum|colproduct|
+---+------+----------+
|  6| 131.0|    2310.0|
|  6| 131.0|    2310.0|
|  4| 115.8|    2120.4|
|  6| 131.4|    2354.0|
|  8| 193.7|    3272.5|
+---+------+----------+
only showing top 5 rows
```

Figure 4.38: New columns in DataFrame

9. Drop both the original numerical columns:

```
data = data.drop('mpg','hp')
data.show(5)
```

```
+---+------+----------+
|cyl|colsum|colproduct|
+---+------+----------+
|  6| 131.0|    2310.0|
|  6| 131.0|    2310.0|
|  4| 115.8|    2120.4|
|  6| 131.4|    2354.0|
|  8| 193.7|    3272.5|
+---+------+----------+
only showing top 5 rows
```

Figure 4.39: New columns in DataFrame after dropping

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

10. Sort the data by categorical column:

```
data = data.orderBy(data.cyl)
data.show(5)
```

The output is as follows:

```
+---+------+----------+
|cyl|colsum|colproduct|
+---+------+----------+
|  4|  98.9|    2203.5|
|  4| 130.4|    2332.6|
|  4| 118.5|    2085.5|
|  4|  82.4|    1580.8|
|  4|  93.3|    1801.8|
+---+------+----------+
only showing top 5 rows
```

Figure 4.40: Sort data by categorical columns

11. Calculate the **mean** of the summation column for each distinct category in the **categorical** variable:

```
data.groupby('cyl').agg({'colsum':'mean'}).show()
```

The output is as follows:

```
+---+------------------+
|cyl|       avg(colsum)|
+---+------------------+
|  4|             109.3|
|  6|142.02857142857144|
|  8|224.31428571428575|
+---+------------------+
```

Figure 4.41: Mean of the summation column

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

12. Filter the rows with values greater than the **mean** of all the **mean** values calculated in the previous step:

```
data.count()#15
cyl_avg = data.groupby('cyl').agg({'colsum':'mean'})
avg = cyl_avg.agg({'avg(colsum)':'mean'}).toPandas().iloc[0,0]
data = data.filter(data.colsum > avg)
data.count()
data.show(5)
```

The output is as follows:

```
+---+------+-----------------+
|cyl|colsum|       colproduct|
+---+------+-----------------+
|  6| 194.7|           3447.5|
|  8| 193.7|           3272.5|
|  8| 196.4|2951.9999999999995|
|  8| 259.3|           3503.5|
|  8| 197.3|           3114.0|
+---+------+-----------------+
only showing top 5 rows
```

Figure 4.42: Mean of all the mean values calculated of the summation column

13. De-duplicate the resultant DataFrame to make sure it has all unique records:

```
data = data.dropDuplicates()
data.count()
```

The output is **15**.

Activity 11: Graphs in Spark

1. Import the required Python libraries in the Jupyter Notebook:

```
import pandas as pd
import os
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Read and show the data from the CSV file using the following command:

```
df = pd.read_csv('mtcars.csv')
df.head()
```

The output is as follows:

	model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Figure 4.43: Auto-mpg DataFrame

3. Visualize the discrete frequency distribution of any continuous numeric variable from your dataset using a histogram:

```
plt.hist(df['mpg'], bins=20)
plt.ylabel('Frequency')
plt.xlabel('Values')
plt.title('Frequency distribution of mpg')
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

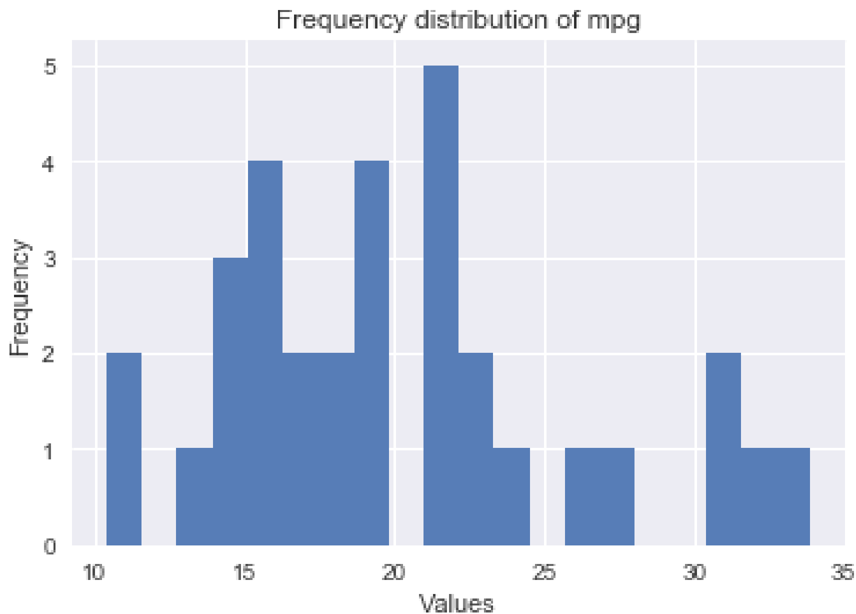


Figure 4.44: Discrete frequency distribution histogram

4. Visualize the percentage share of the categories in the dataset using a pie chart:

```
## Calculate count of records for each gear
data = pd.DataFrame([[3,4,5],df['gear'].value_counts().tolist()]).T
data.columns = ['gear','gear_counts']

## Visualising percentage contribution of each gear in data using pie
chart
plt.pie(data.gear_counts, labels=data.gear, startangle=90,
autopct='%.1f%%')
plt.title('Percentage contribution of each gear')
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

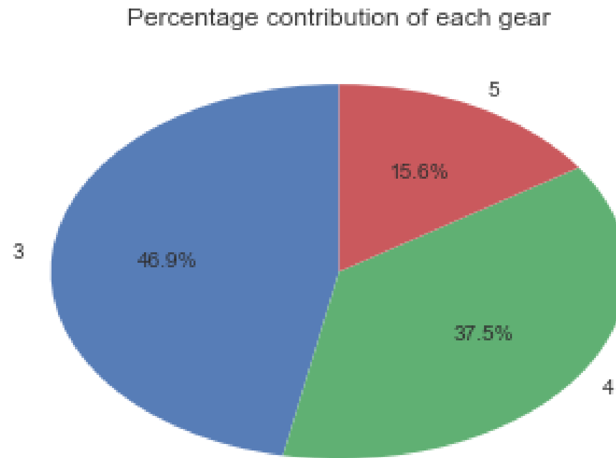


Figure 4.45: Percentage share of the categories using pie chart

5. Plot the distribution of a continuous variable across the categories of a categorical variable using a boxplot:

```
sns.boxplot(x = 'gear', y = 'disp', data = df)
plt.show()
```

The output is as follows:

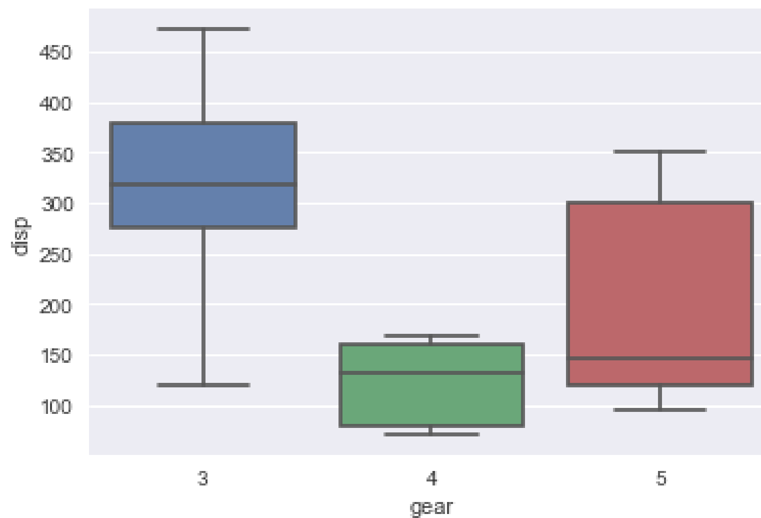


Figure 4.46: Distribution of a continuous using boxplot

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Visualize the values of a continuous numeric variable using a line chart:

```
data = df[['hp']]
data.plot(linestyle='-')
plt.title('Line Chart for hp')
plt.ylabel('Values')
plt.xlabel('Row number')
plt.show()
```

The output is as follows:

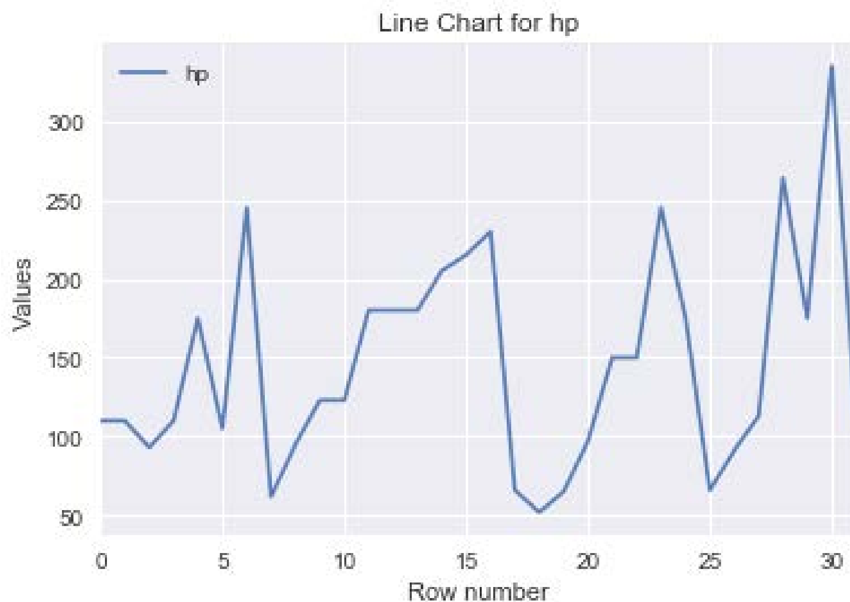


Figure 4.47: Continuous numeric variable using a line chart

7. Plot the values of multiple continuous numeric variables on the same line chart:

```
data = df[['hp','disp', 'mpg']]
data.plot(linestyle='-')
plt.title('Line Chart for hp, disp & mpg')
plt.ylabel('Values')
plt.xlabel('Row number')
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

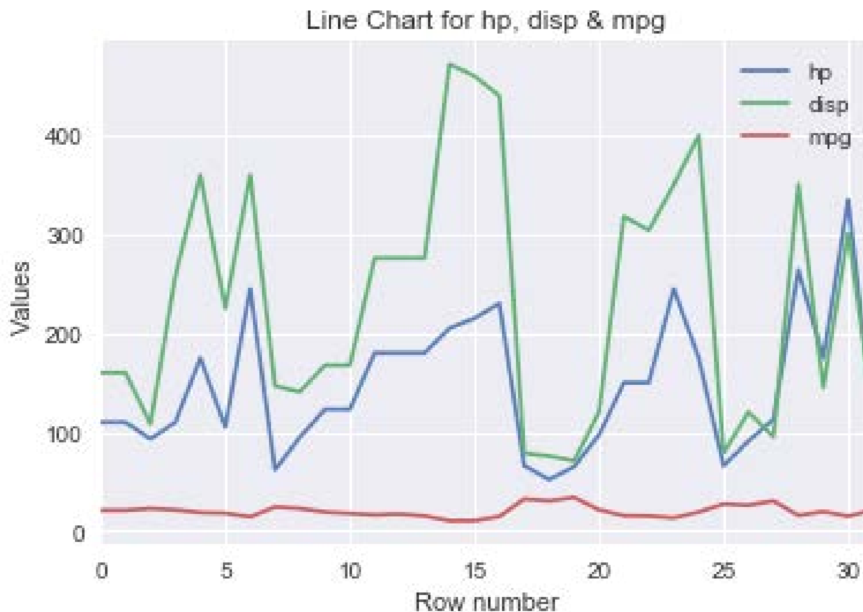


Figure 4.48: Multiple continuous numeric variables

Chapter 05: Missing Value Handling and Correlation Analysis in Spark

Activity 12: Missing Value Handling and Correlation Analysis with PySpark DataFrames

1. Import the required libraries and modules in the Jupyter notebook, as illustrated here:

```
import findspark
findspark.init()
import pyspark
import random
```

2. Set up the **SparkContext** with the help of the following command in the Jupyter notebook:

```
sc = pyspark.SparkContext(appName = "chapter5")
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

3. Similarly, set up the **SQLContext** in the notebook:

```
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

4. Now, read the CSV data into a Spark object using the following command:

```
df = sqlc.read.format('com.databricks.spark.csv').options(header = 'true',
inferschema = 'true').load('iris.csv')
df.show(5)
```

The output is as follows:

```
+-----------+----------+-----------+----------+-------+
|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-----------+----------+-----------+----------+-------+
|        5.1|       3.5|        1.4|       0.2| setosa|
|        4.9|       3.0|        1.4|       0.2| setosa|
|        4.7|       3.2|        1.3|       0.2| setosa|
|        4.6|       3.1|        1.5|       0.2| setosa|
|        5.0|       3.6|        1.4|       0.2| setosa|
+-----------+----------+-----------+----------+-------+
only showing top 5 rows
```

Figure 5.14: Iris DataFrame, reading the CSV data into a Spark object

5. Fill in the missing values in the **Sepallength** column with the column's mean.
6. First, calculate the mean of the **Sepallength** column using the following command:

```
from pyspark.sql.functions import mean
avg_sl = df.select(mean('Sepallength')).toPandas()['avg(Sepallength)']
```

7. Now, impute the missing values in the **Sepallength** column with the column's mean, as illustrated here:

```
y = df
y = y.na.fill(float(avg_sl),['Sepallength'])
y.describe().show(1)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
+-------+-----------+----------+-----------+----------+-------+
|summary|Sepallength|Sepalwidth|Petallength|Petalwidth|Species|
+-------+-----------+----------+-----------+----------+-------+
|  count|        150|       150|        149|       150|    150|
+-------+-----------+----------+-----------+----------+-------+
only showing top 1 row
```

Figure 5.15: Iris DataFrame

8. Compute the correlation matrix for the dataset. Make sure to import the required modules, as shown here:

```
from pyspark.mllib.stat import Statistics
import pandas as pd
```

9. Now, fill the missing values in the DataFrame before computing the correlation:

```
z = y.fillna(1)
```

10. Next, remove the **`String`** columns from the PySpark DataFrame, as illustrated here:

```
a = z.drop('Species')
features = a.rdd.map(lambda row: row[0:])
```

11. Now, compute the correlation matrix in Spark:

```
correlation_matrix = Statistics.corr(features, method="pearson")
```

12. Next, convert the correlation matrix into a pandas DataFrame using the following command:

```
correlation_df = pd.DataFrame(correlation_matrix)
correlation_df.index, correlation_df.columns = a.columns, a.columns
correlation_df
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

	Sepallength	Sepalwidth	Petallength	Petalwidth
Sepallength	1.000000	-0.113841	0.861480	0.807310
Sepalwidth	-0.113841	1.000000	-0.427570	-0.366126
Petallength	0.861480	-0.427570	1.000000	0.962741
Petalwidth	0.807310	-0.366126	0.962741	1.000000

Figure 5.16: Convert the correlation matrix into a pandas DataFrame

13. Plot the variable pairs showing strong positive correlation and fit a linear line on them.
14. First, load the data from the Spark DataFrame into a pandas DataFrame:

```
import pandas as pd
dat = y.toPandas()
type(dat)
```

The output is as follows:

```
pandas.core.frame.DataFrame
```

15. Next, load the required modules and plotting data using the following commands:

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.lmplot(x = "Sepallength", y = "Petallength", data = dat)
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

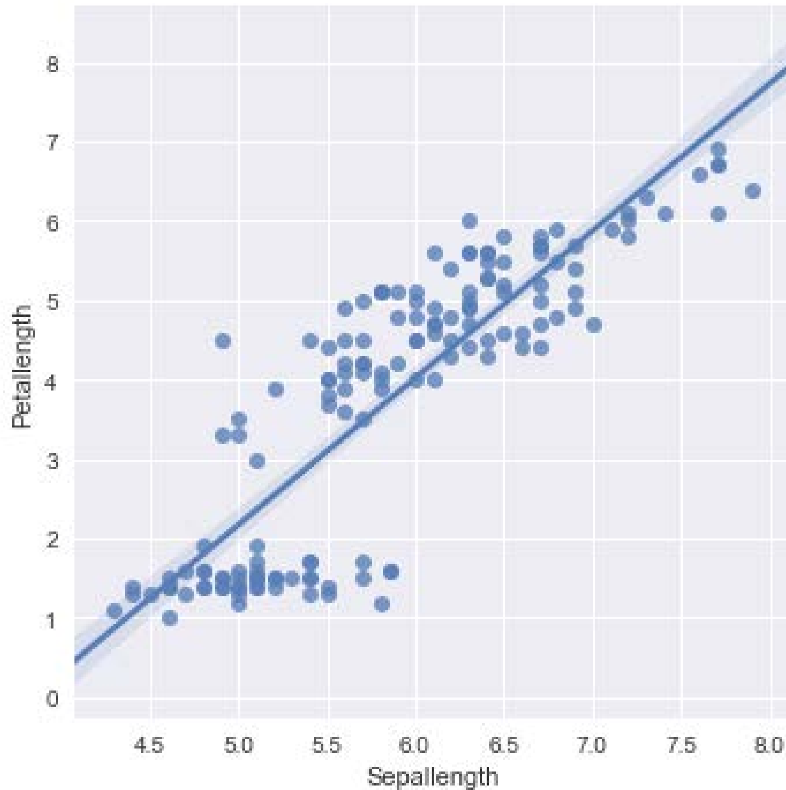


Figure 5.17: Seaborn plot for x = "Sepallength", y = "Petallength"

16. Plot the graph so that **x** equals **Sepallength**, and **y** equals **Petalwidth**:

```
import seaborn as sns
sns.lmplot(x = "Sepallength", y = "Petalwidth", data = dat)
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

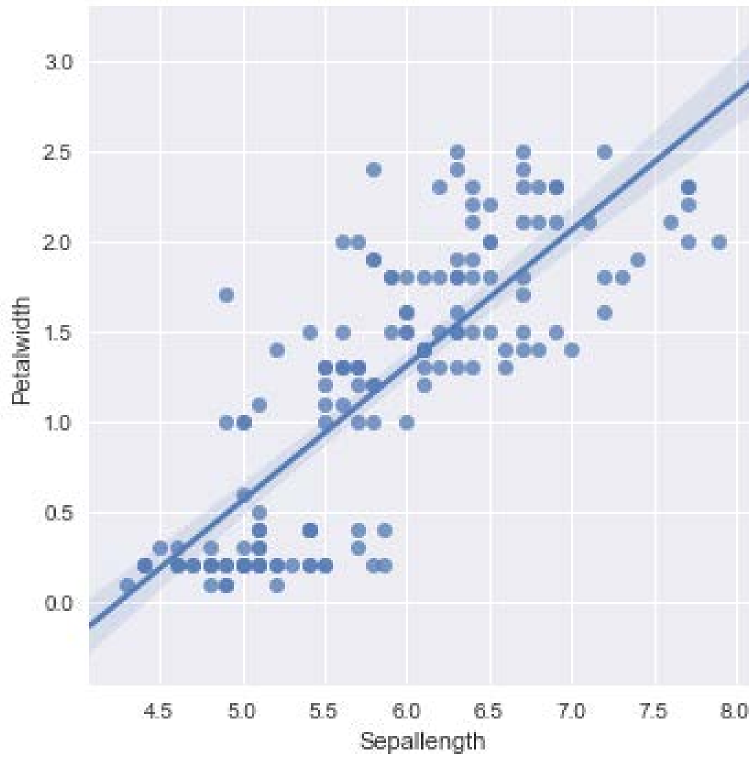


Figure 5.18: Seaborn plot for x = "Sepallength", y = "Petalwidth"

17. Plot the graph so that **x** equals **Petalwidth** and **y** equals **Petalwidth**:

```
sns.lmplot(x = "Petallength", y = "Petalwidth", data = dat)
plt.show()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

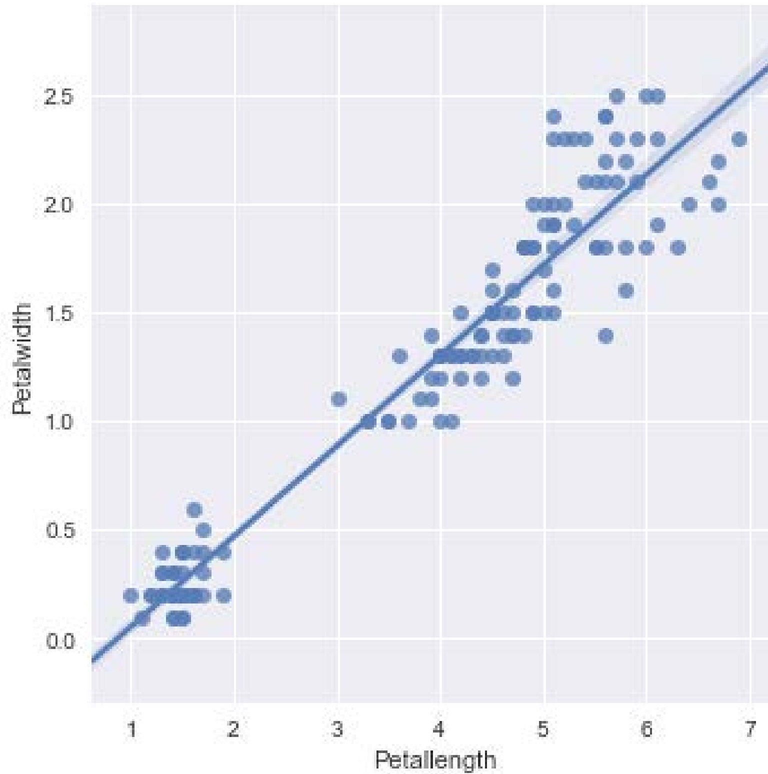


Figure 5.19: Seaborn plot for x = "Petallength", y = "Petalwidth"

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Chapter 6: Business Process Definition and Exploratory Data Analysis

Activity 13: Carry Out Mapping to Gaussian Distribution of Numeric Features from the Given Data

1. Download the **bank.csv**. Now, use the following commands to read the data from it:

```
import numpy as np
import pandas as pd
import seaborn as sns
import time
import re
import os
import matplotlib.pyplot as plt
sns.set(style="ticks")

# import libraries required for preprocessing
import sklearn as sk
from scipy import stats
from sklearn import preprocessing

# set the working directory to the following
os.chdir("/Users/svk/Desktop/packt_exercises")

# read the downloaded input data (marketing data)
df = pd.read_csv('bank.csv', sep=';')
```

2. Identify the numeric data from the DataFrame. The data can be categorized according to its type, such as categorical, numeric (float, integer), date, and so on. We identify numeric data here because we can only carry out normalization on numeric data:

```
numeric_df = df._get_numeric_data()
numeric_df.head()
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
# lets segment the data to numeric and categorical and carry out distribution transformation on the numeric data

# for getting numerical data from the raw data
numeric_df = df._get_numeric_data()
numeric_df.head()
```

	age	balance	day	duration	campaign	pdays	previous
0	30	1787	19	79	1	-1	0
1	33	4789	11	220	1	339	4
2	35	1350	16	185	1	330	1
3	30	1476	3	199	4	-1	0
4	59	0	5	226	1	-1	0

Figure 6.12: DataFrame

3. Carry out a normality test and identify the features that have a non-normal distribution:

```
numeric_df_array = np.array(numeric_df) # converting to numpy arrays for
more efficient computation

loop_c = -1
col_for_normalization = list()

for column in numeric_df_array.T:
    loop_c+=1
    x = column
    k2, p = stats.normaltest(x)
    alpha = 0.001
    print("p = {:g}".format(p))

    # rules for printing the normality output
    if p < alpha:
        test_result = "non_normal_distr"
        col_for_normalization.append((loop_c)) # applicable if yeo-johnson
is used

        #if min(x) > 0: # applicable if box-cox is used
            #col_for_normalization.append((loop_c)) # applicable if
box-cox is used
        print("The null hypothesis can be rejected: non-normal
distribution")

    else:
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
            test_result = "normal_distr"
            print("The null hypothesis cannot be rejected: normal
    distribution")
```

The output is as follows:

```
p = 1.98749e-70
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 3.08647e-278
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
```

Figure 6.13: Normality test and identify the features

> **Note**
>
> The normality test conducted here is based on D'Agostino and Pearson's test (https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.normaltest.html), which combines skew and kurtosis to identify how close the distribution of the features is to a Gaussian distribution. In this test, if the p-value is less than the set alpha value, then the null hypothesis is rejected, and the feature does not have a normal distribution. Here, we look into each column using a loop function and identify the distribution of each feature.

4. Plot the probability density of the features to visually analyze their distribution:

```
columns_to_normalize = numeric_df[numeric_df.columns[col_for_
normalization]]
names_col = list(columns_to_normalize)

# density plots of the features to check the normality
columns_to_normalize.plot.kde(bw_method=3)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The density plot of the features to check the normality is as follows:

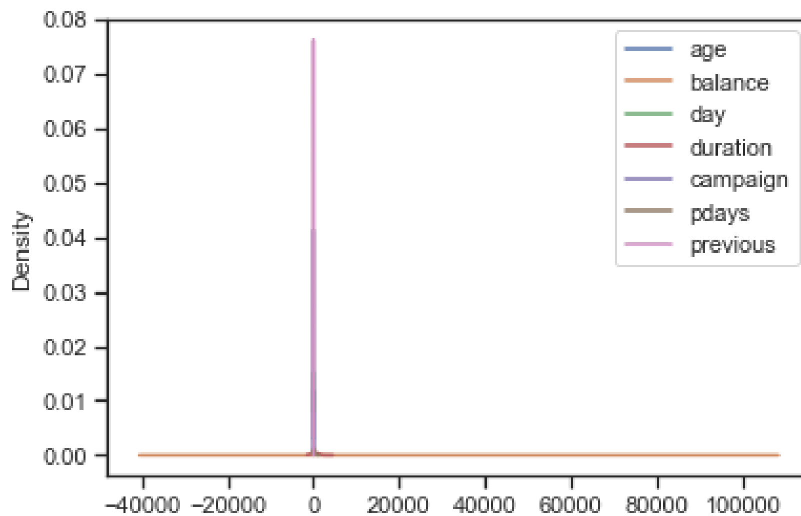


Figure 6.14: Plot of features

> **Note**
>
> Multiple variables' density plots are shown in the previous graph. The distribution of the features in the graph can be seen with a high positive kurtosis, which is not a normal distribution.

5. Prepare the power transformation model and carry out transformations on the identified features to convert them to normal distribution based on the **box-cox** or **yeo-johnson** method:

```
pt = preprocessing.PowerTransformer(method='yeo-johnson',
standardize=True, copy=True)
normalized_columns = pt.fit_transform(columns_to_normalize)
normalized_columns = pd.DataFrame(normalized_columns, columns=names_col)
```

In the previous commands, we prepare the power transformation model and apply it to the data of selected features.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

6. Plot the probability density of the features again after the transformations to visually analyze the distribution of the features:

```
normalized_columns.plot.kde(bw_method=3)
```

The output is as follows:

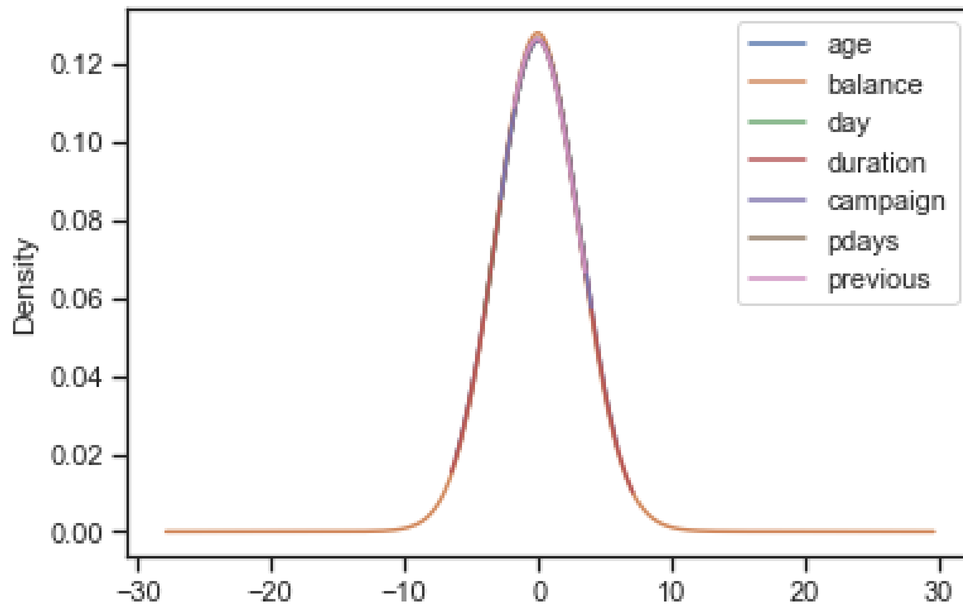


Figure 6.15: Plot of features

Chapter 07: Reproducibility in Big Data Analysis

Activity 14: Test normality of data attributes (columns) and carry out Gaussian normalization of non-normally distributed attributes

1. Import the required libraries and packages in the Jupyter notebook:

```
import numpy as np
import pandas as pd
import seaborn as sns
import time
import re
import os
import matplotlib.pyplot as plt
sns.set(style="ticks")
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

2. Now, import the libraries required for preprocessing:

```
import sklearn as sk
from scipy import stats
from sklearn import preprocessing
```

3. Set the working directory using the following command:

```
os.chdir("/Users/svk/Desktop/packt_exercises")
```

4. Now, import the dataset into the Spark object:

```
df = pd.read_csv('bank.csv', sep=';')
```

5. Identify the target variable in the data:

```
DV = 'y'
df[DV]= df[DV].astype('category')
df[DV] = df[DV].cat.codes
```

6. Generate training and testing data using the following command:

```
msk = np.random.rand(len(df)) < 0.8
train = df[msk]
test = df[~msk]
```

7. Create the **Y** and **X** data, as illustrated here:

```
# selecting the target variable (dependent variable) as y
y_train = train[DV]
```

8. Drop the **DV** or **y** using the **drop** command:

```
train = train.drop(columns=[DV])
train.head()
```

The output is as follows:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	p
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1	
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339	
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330	
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1	
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	may	226	1	-1	

Figure 7.22: Bank dataset

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

9. Segment the data numerically and categorically and perform distribution transformation on the numeric data:

```
numeric_df = train._get_numeric_data()
```

Perform data preprocessing on the data.

10. Now, create a **loop** to identify the columns with a non-normal distribution using the following command (converting to NumPy arrays for more efficient computation):

```
numeric_df_array = np.array(numeric_df)
loop_c = -1
col_for_normalization = list()

for column in numeric_df_array.T:
    loop_c+=1
    x = column
    k2, p = stats.normaltest(x)
    alpha = 0.001
    print("p = {:g}".format(p))

    # rules for printing the normality output
    if p < alpha:
        test_result = "non_normal_distr"
        col_for_normalization.append((loop_c)) # applicable if yeo-johnson 
is used

        #if min(x) > 0: # applicable if box-cox is used
            #col_for_normalization.append((loop_c)) # applicable if 
box-cox is used
        print("The null hypothesis can be rejected: non-normal 
distribution")

    else:
        test_result = "normal_distr"
        print("The null hypothesis cannot be rejected: normal 
distribution")
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

```
p = 6.57189e-54
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 2.63426e-215
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
```

Figure 7.23: Identifying the columns with a non-linear distribution

11. Create a **PowerTransformer** based transformation (**box-cox**):

```
pt = preprocessing.PowerTransformer(method='yeo-johnson',
standardize=True, copy=True)
```

> **Note**
>
> **box-cox** can handle only positive values.

12. Apply the power transformation model on the data. Select the columns to normalize:

```
columns_to_normalize = numeric_df[numeric_df.columns[col_for_
normalization]]
names_col = list(columns_to_normalize)
```

13. Create a density plot to check the normality:

```
columns_to_normalize.plot.kde(bw_method=3)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

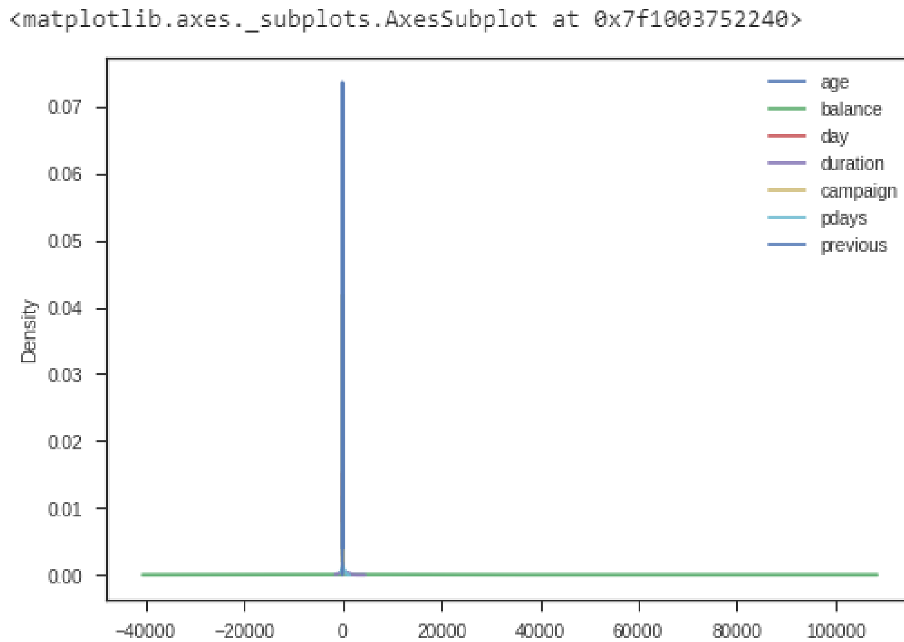


Figure 7.24: Density plot to check the normality

14. Now, transform the columns to a normal distribution using the following command:

```
normalized_columns = pt.fit_transform(columns_to_normalize)
normalized_columns = pd.DataFrame(normalized_columns, columns=names_col)
```

15. Again, create a density plot to check the normality:

```
normalized_columns.plot.kde(bw_method=3)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The output is as follows:

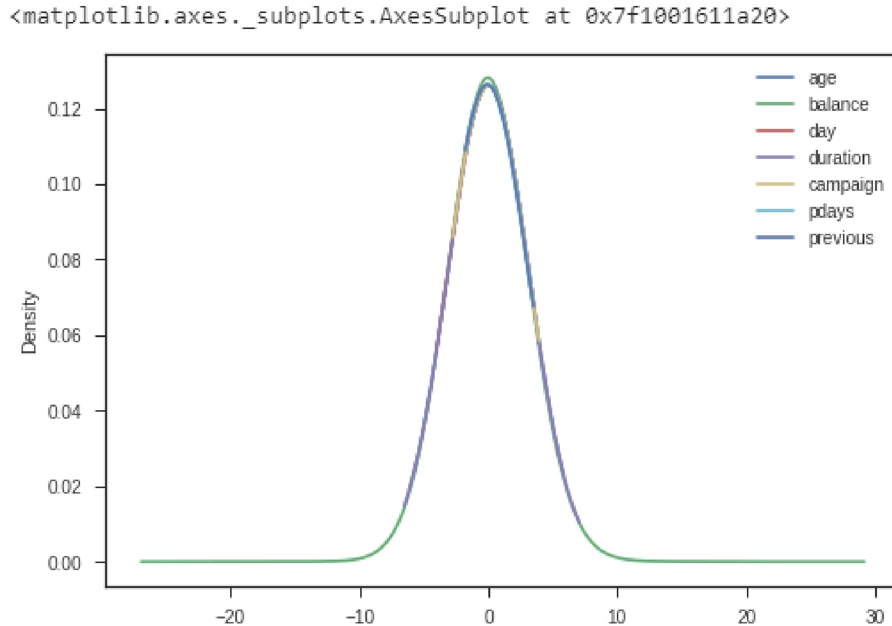


Figure 7.25: Another density plot to check the normality

16. Use a **loop** to identify the columns with non-normal distribution on the transformed data:

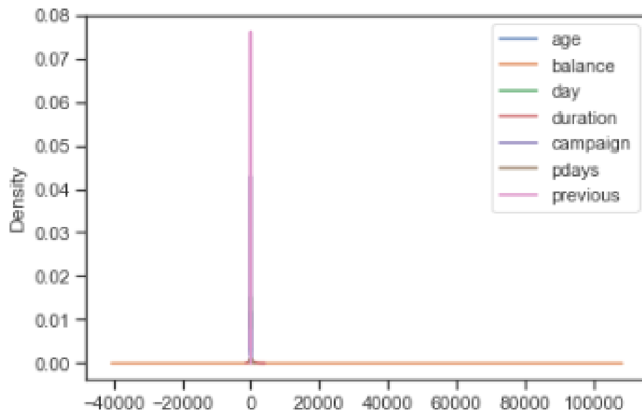
```
numeric_df_array = np.array(normalized_columns)
loop_c = -1

for column in numeric_df_array.T:
    loop_c+=1
    x = column
    k2, p = stats.normaltest(x)
    alpha = 0.001
    print("p = {:g}".format(p))

    # rules for printing the normality output
    if p < alpha:
        test_result = "non_normal_distr"
        print("The null hypothesis can be rejected: non-normal
distribution")

    else:
        test_result = "normal_distr"
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

```
        print("The null hypothesis cannot be rejected: normal
distribution")
```

The output is as follows:

```
p = 1.05883e-20
The null hypothesis can be rejected: non-normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 3.06698e-155
The null hypothesis can be rejected: non-normal distribution
p = 0.00448565
The null hypothesis cannot be rejected: normal distribution
p = 0
The null hypothesis can be rejected: non-normal distribution
p = 2.37302e-191
The null hypothesis can be rejected: non-normal distribution
p = 1.70855e-191
The null hypothesis can be rejected: non-normal distribution
```



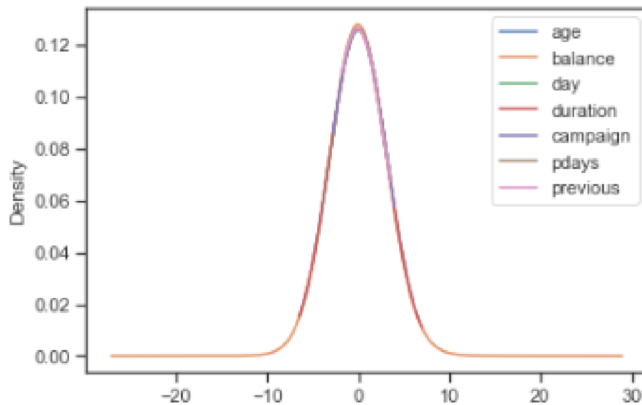


Figure 7.26: Power transformation model to data

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

17. Bind the normalized and non-normalized columns. Select the columns not to normalize:

```
columns_to_notnormalize = numeric_df
columns_to_notnormalize.drop(columns_to_notnormalize.columns[col_for_
normalization], axis=1, inplace=True)
```

18. Use the following command to bind both the non-normalized and normalized columns:

```
numeric_df_normalized = pd.concat([columns_to_notnormalize.reset_
index(drop=True), normalized_columns], axis=1)
numeric_df_normalized
```

	age	balance	day	duration	campaign	pdays	previous
0	-1.131949	0.275589	0.430558	-0.912349	-1.107958	-0.477714	-0.477734
1	-0.734064	1.192878	-0.529802	0.145328	-1.107958	2.117009	2.153145
2	-0.492714	0.125717	0.086950	-0.044964	-1.107958	2.116381	1.999758
3	-1.131949	0.169655	-1.726794	0.034568	1.085510	-0.477714	-0.477734
4	1.508851	-0.441275	-1.387705	0.175309	-1.107958	-0.477714	-0.477734
5	-0.492714	-0.095740	0.865685	-0.333744	0.133678	2.098513	2.147552
6	-0.378316	-0.277298	-0.152213	0.648433	-1.107958	2.116381	2.126717
7	-0.057331	-0.352990	-1.231266	-0.261967	0.133678	-0.477714	-0.477734
8	0.140189	-0.316857	-0.152213	-1.216755	0.133678	-0.477714	-0.477734
9	0.326147	-0.625045	0.203356	0.547565	-1.107958	2.092070	2.126717
10	-0.057331	2.422831	0.541599	0.389145	-1.107958	-0.477714	-0.477734

Figure 7.27: Non-normalized and normalized columns

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

Chapter 08: Creating a Full Analysis Report

Activity 15: Generating Visualization Using Plotly

1. Import all the required libraries and packages into the Jupyter notebook. Make sure to read the data from **bank.csv** into the Spark DataFrame.
2. Import the libraries for Plotly, as illustrated here:

```
import plotly.graph_objs as go
from plotly.plotly import iplot
import plotly as py
```

3. Now, for visualization in Plotly, we need to initiate an offline session. Use the following command (requires version >= 1.9.0):

```
from plotly import __version__
from plotly.offline import download_plotlyjs, init_notebook_mode, plot,
iplot
print(__version__)
```

4. Now Plotly is initiated offline. Use the following command to start a Plotly notebook:

```
import plotly.plotly as py
import plotly.graph_objs as go

init_notebook_mode(connected=True)
```

After starting the Plotly notebook, we can use Plotly to generate many types of graphs, such as a bar graph, a boxplot, or a scatter plot, and convert the entire output into a user interface or an app that is supported by Python's Flask framework.

5. Now, plot each graph using Plotly:

Bar graph:

```
df = pd.read_csv('bank.csv', sep=';')
data = [go.Bar(x=df.y,
            y=df.balance)]

py.iplot(data)
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The bar graph is as follows:

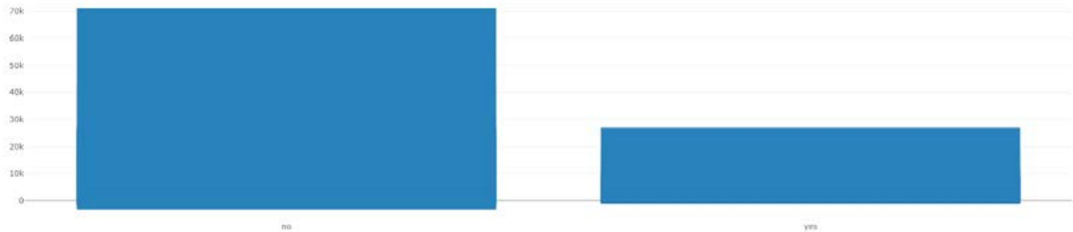


Figure 8.18: Bar graph

Scatter plot:

```
py.iplot([go.Histogram2dContour(x=df.balance, y=df.age,
contours=dict(coloring='heatmap')),
       go.Scatter(x=df.balance, y=df.age, mode='markers',
marker=dict(color='red', size=8, opacity=0.3))], show_link=False)
```

The scatter plot is as follows:

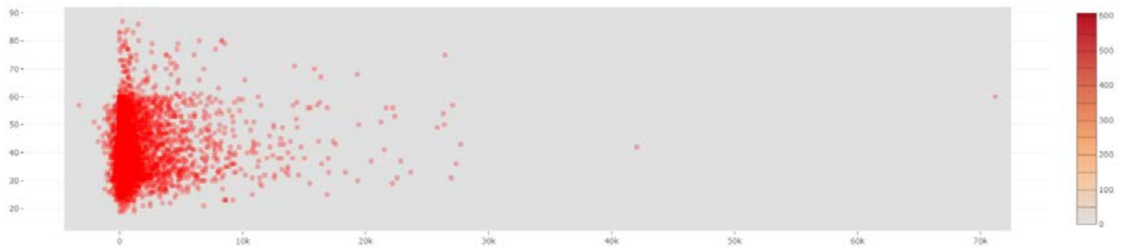


Figure 8.19: Scatter plot

Boxplot:

```
plot1 = go.Box(
    y=df.age,
    name = 'age of the customers',
    marker = dict(
        color = 'rgb(12, 12, 140)',
    )
)
py.iplot([plot1])
```

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

The boxplot is as follows:

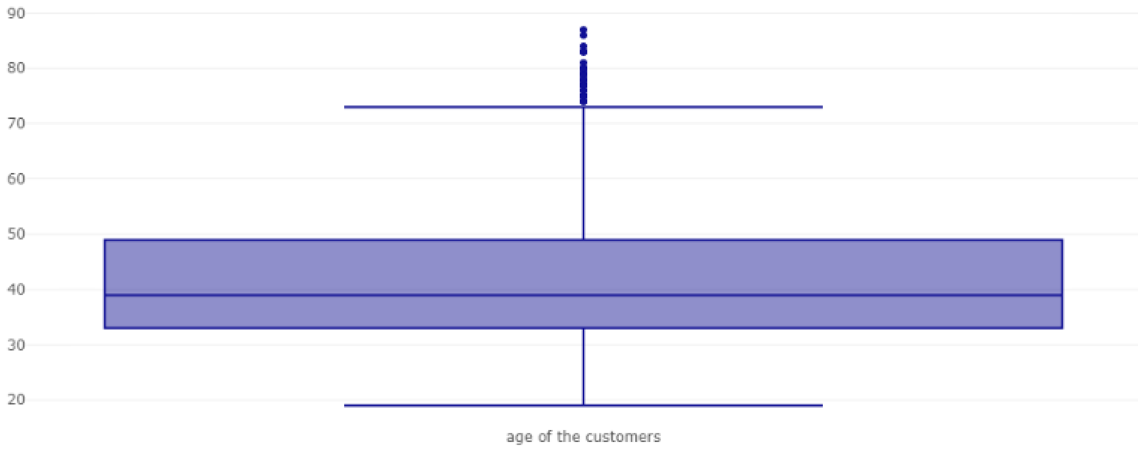
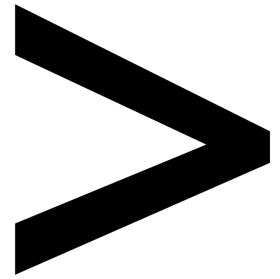


Figure 8.20: Boxplot

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

A

aggregate: 24, 26, 31, 34, 108
apache: 73, 76, 82
applymap: 20, 23, 26-28
appname: 78, 81, 183, 185, 190-191
arithmetic: 8, 177
arrays: 10, 177
astype: 21-23, 28, 149
auto-mpg: 46-48, 50-52, 55-56, 62, 66-67

B

backend: 10, 92
background: 62
barplot: 108, 140, 143-144, 150
boolean: 104
borutapy: 149, 194
box-cox: 154, 171, 193
boxplot: 42, 54-57, 68, 110-111, 113, 137, 196
brottli: 83

C

campaign: 138, 146
cartesian: 42
cassandra: 77
chi-square: 138, 147
coherent: 34, 131
configured: 38, 58, 60, 74, 81
connector: 85
consensus: 2
console: 3-4, 6, 98
context: 72, 107, 118, 184
coordinate: 37, 39
copy-paste: 3
crosstab: 188

D

database: 81-82, 85, 152, 184
databricks: 96, 119
datanode: 74
datatype: 186
datetime: 11, 21-23, 28, 52, 56
debugging: 9, 161, 176
domain: 134, 152
dtypes: 22-23

E

encode: 28
encoding: 28, 153
endpoint: 81
enumerates: 34

F

facecolor: 65
figsize: 61, 64, 146
findspark: 118
flatmap: 86
floats: 147
fontsize: 147
framework: 89

G

gaussian: 153-155
github: 6, 12, 17
graphx: 76, 89
grayscale: 62
groupby: 11-12, 24-26, 46, 59-60, 67, 80, 106, 108, 146

H

hadoop: 68, 71-76, 82, 89, 197
hands-on: 6, 114
heatmap: 147-148
histogram: 42, 51-53, 66, 68, 112, 141-143, 145-146, 167

I

immutable: 78, 92
ingest: 12, 31, 50, 66-67, 160
insert: 6
integer: 21, 47, 52, 56, 154
intuitive: 100, 132
ipython: 1-5, 9, 11, 34, 77, 131, 157
isnull: 120-121, 123-124
isotope: 18
issues: 20, 82, 134
italic: 162, 164
italicized: 164
italics: 164
iterate: 9
iterative: 9
itself: 38, 92

K

kaggle: 112
kdeplot: 110
kernel: 5-6, 110
kubernetes: 76
kurtosis: 110

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

L

lambda: 20, 23, 27-28, 86-87, 126
linear: 10, 41, 50, 108-110, 125-127, 154, 193
lineplot: 45
linspace: 35, 39-41
literature: 34
little: 6, 93-94
lmplot: 109
localhost: 6

M

markdown: 5-6, 8, 162, 164-165, 171
master: 12, 17, 46, 62, 86, 167, 173, 185
metastore: 77
ml-bank: 183, 185, 190-191
mojave: 92
mtcars: 112
mydatabase: 81
myjsonfile: 81
mypassword: 81, 85
mytable: 81-82

N

namenode: 74
narratives: 9
ndarray: 10

O

opencv: 161
opendata: 17, 22, 25-26
orderby: 105

P

parquet: 13, 26, 28, 71, 77, 80, 82-85, 89, 93, 96
pearson: 124-126, 129, 138
petalwidth: 128-129
pictograms: 38
pinpoint: 161
plotly: 34, 181, 194-197
plotlyjs: 195
portable: 74
portuguese: 138, 189
postgre: 81, 85
postgresql: 77, 81-82, 85
profiling: 166-168, 170, 183
pyarrow: 13
pydata: 11, 41
pyplot: 11, 22, 25, 35, 38, 40, 44, 50-51, 55, 58, 62, 64, 66, 68, 108, 139, 147, 183
pyspark: 77-78, 81, 92-94, 96, 114, 117-127, 129, 179, 182-185, 189-191

R

regression: 41, 122, 154-155, 193
resolved: 161, 164

S

sandbox: 173
savefig: 64-66
scalable: 92
scatter: 29-30, 42, 48-50, 61-65, 68, 137, 196
schema: 82, 93-94, 97-98, 100-101, 183-185
seaborn: 1, 11, 22, 25, 31, 33, 41, 44-45, 49, 51, 55-58, 62-63, 65, 78, 108, 110, 139, 141-142, 147, 183, 197
semicolon: 28
sensor-: 137
sentiment: 133
sepalwidth: 99, 101-102, 106, 109-111, 125
setosa: 95, 104
sklearn: 149, 183
snappy: 83, 85
snippet: 93, 106, 159, 164, 171
socrata: 12, 17, 22, 25-26
sortby: 87
spark-csv: 96-97, 119
sqlcontext: 93-94, 119, 126
statistics: 10, 21, 24, 34, 41, 78, 97-100, 114, 121, 125-126, 185, 187
statlib: 46
stopwords: 88
subset: 14, 19, 191

T

tables: 11, 68, 72, 93, 153
tabular: 2, 11, 85, 97
tabulation: 34
tensorflow: 161

U

up-sell: 133
username: 195

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

V

values: 8-9, 14-15, 17, 19-21, 24-26, 28, 37-38, 41, 44, 51, 54, 79, 82, 84-85, 103, 106-107, 113-114, 117-127, 129, 140, 143-144, 147-150, 155, 167, 171, 174-175, 185
vector: 155, 193
vendor: 132
verbose: 41, 149
violin: 41
virginica: 95
virtualenv: 1

W

wallet: 134
whitespace: 52, 55, 62
workflow: 9, 34, 133-134, 152, 158-160, 170, 173
wrapper: 149

X

x-axis: 29, 37, 40, 42-43, 58, 61
xlabel: 58, 108, 145
xticks: 147

Y

y-axis: 29, 38, 42, 58, 61
ylabel: 58, 108, 145
yticks: 147

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use

EBSCOhost - printed on 2/9/2023 10:53 AM via . All use subject to https://www.ebsco.com/terms-of-use