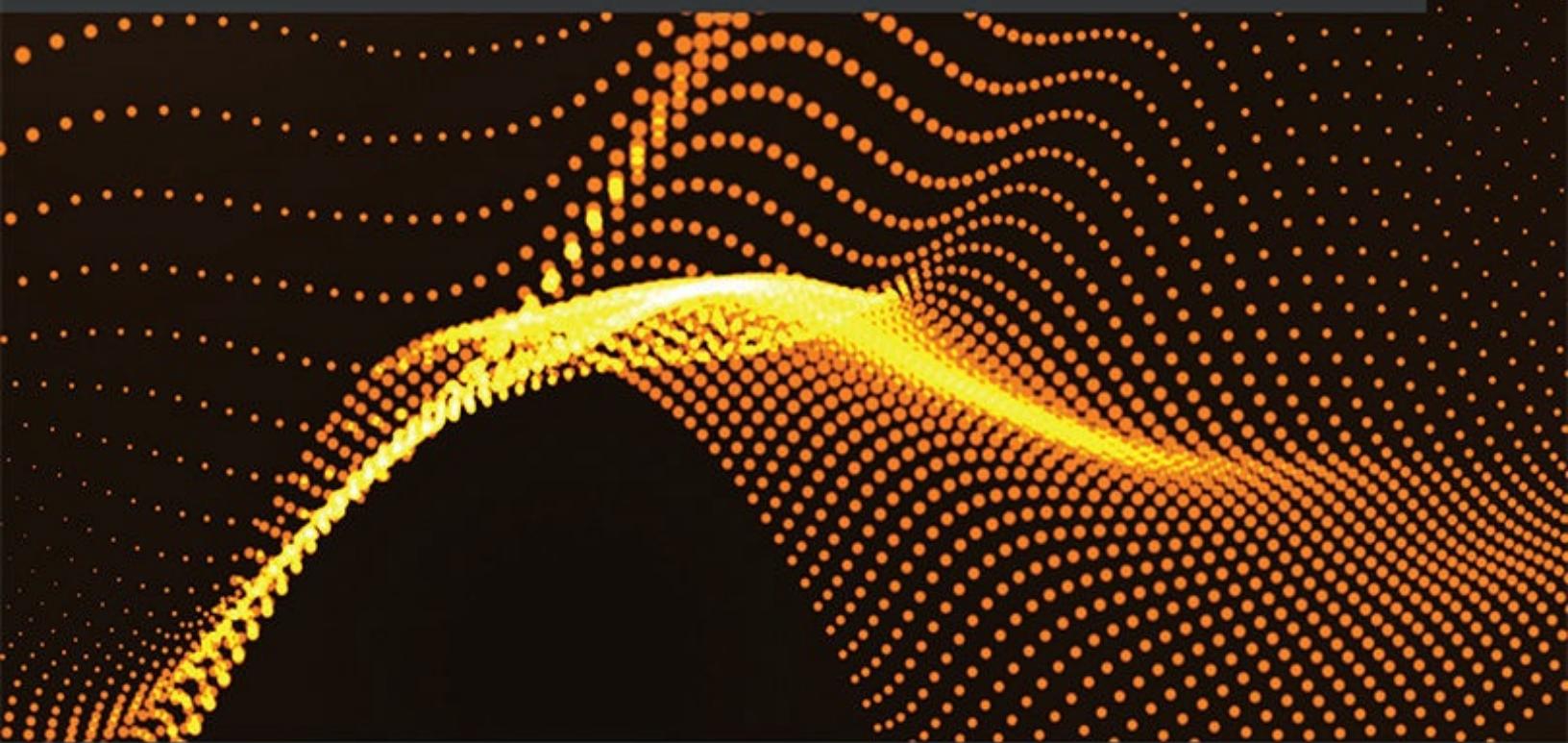


Extending Power BI with Python and R

Ingest, transform, enrich, and visualize data using
the power of analytical languages



Luca Zavarella
Foreword by Francesca Lazzeri



Extending Power BI with Python and R

Copyright © 2021 Packt Publishing

This is an Early Access product. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the content and extracts of this book may evolve as it is being developed to ensure it is up-to-date.

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.

The information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

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

Early Access Publication: Extending Power BI with Python and R

Early Access Production Reference: B17081

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80107-820-7

www.packt.com

Table of Contents

1. [Extending Power BI with Python and R: Ingest, transform, enrich and visualize using the power of analytical languages](#)
2. [1 Where and How to Use R and Python Scripts in Power BI](#)
 1. [Technical requirements](#)
 2. [Injecting R or Python scripts into Power BI](#)
 1. [Data loading](#)
 2. [Data transformation](#)
 3. [Data visualization](#)
 3. [Using R and Python to interact with your data](#)
 4. [R and Python limitations on Power BI products](#)
 5. [Summary](#)
3. [2 Configuring R with Power BI](#)
 1. [Technical requirements](#)
 2. [The available R engines](#)
 1. [The CRAN R distribution](#)
 2. [The Microsoft R Open distribution and MRAN](#)
 3. [Microsoft R Client](#)
 3. [Choosing an R engine to install](#)
 1. [The R engines used by Power BI](#)
 2. [Installing the suggested R engines](#)
 4. [Installing an IDE for R development](#)
 1. [Installing RStudio](#)
 5. [Configuring Power BI Desktop to work with R](#)
 6. [Configuring the Power BI service to work with R](#)
 1. [Installing the on-premises data gateway in personal mode](#)
 2. [Sharing reports that use R scripts in the Power BI service](#)
 7. [R visuals limitations](#)
 8. [Summary](#)
4. [3 Configuring Python with Power BI](#)
 1. [Technical requirements](#)
 2. [The available Python engines](#)
 3. [Choosing a Python engine to install](#)
 1. [The Python engines used by Power BI](#)

2. [Installing the suggested Python engines](#)
4. [Installing an IDE for Python development](#)
 1. [Configuring Python with RStudio](#)
 2. [Configuring Python with Visual Studio Code](#)
5. [Configuring Power BI Desktop to work with Python](#)
6. [Configuring the Power BI service to work with R](#)
 1. [Sharing reports that use Python scripts in the Power BI service](#)
 7. [Limitations of Python visuals](#)
 8. [Summary](#)
5. [4 Importing Unhandled Data Objects](#)
 1. [Technical requirements](#)
 2. [Importing RDS files in R](#)
 1. [A brief introduction to Tidyverse](#)
 2. [Creating a serialized R object](#)
 3. [Using an RDS file in Power BI](#)
 3. [Importing PKL files in Python](#)
 1. [A very short introduction to the PyData world](#)
 2. [Creating a serialized Python object](#)
 3. [Using a PKL file in Power BI](#)
 4. [Summary](#)
 5. [References](#)
6. [5 Using Regular Expressions in Power BI](#)
 1. [Technical requirements](#)
 2. [A brief introduction to regexes](#)
 1. [The basics of regexes](#)
 2. [Checking the validity of email addresses](#)
 3. [Checking the validity of dates](#)
 3. [Validating data using regex in Power BI](#)
 1. [Using regex in Power BI to validate emails with Python](#)
 2. [Using regex in Power BI to validate emails with R](#)
 3. [Using regex in Power BI to validate dates with Python](#)
 4. [Using regex in Power BI to validate dates with R](#)
 4. [Loading complex log files using regex in Power BI](#)
 1. [Apache access logs](#)
 2. [Importing Apache access logs in Power BI with Python](#)
 3. [Importing Apache access logs in Power BI with R](#)

5. [Extracting values from text using regex in Power BI](#)
 1. [One regex to rule them all](#)
 2. [Using regex in Power BI to extract values with Python](#)
 3. [Using regex in Power BI to extract values with R](#)
6. [Summary](#)
7. [References](#)
7. [6 Anonymizing and Pseudonymizing your Data in Power BI](#)
 1. [Technical requirements](#)
 2. [De-identifying data](#)
 1. [De-identification techniques](#)
 2. [Understanding pseudonymization](#)
 3. [What is anonymization?](#)
 3. [Anonymizing data in Power BI](#)
 1. [Anonymizing data using Python](#)
 2. [Anonymizing data using R](#)
 4. [Pseudonymizing data in Power BI](#)
 1. [Pseudonymizing data using Python](#)
 2. [Pseudonymizing data using R](#)
 5. [Summary](#)
 6. [References](#)
8. [7 Logging Data from Power BI to External Sources](#)
 1. [Technical requirements](#)
 2. [Logging to CSV files](#)
 1. [Logging to CSV files with Python](#)
 2. [Logging to CSV files with R](#)
 3. [Logging to Excel files](#)
 1. [Logging to Excel files with Python](#)
 2. [Logging to Excel files with R](#)
 4. [Logging to an Azure SQL Server](#)
 1. [Installing SQL Server Express](#)
 2. [Creating an Azure SQL database](#)
 3. [Logging to an Azure SQL server with Python](#)
 4. [Logging to an Azure SQL server with R](#)
 5. [Summary](#)
 6. [References](#)
9. [8 Loading Large Datasets beyond the Available RAM in Power BI](#)
 1. [Technical requirements](#)

2. [A typical analytic scenario using large datasets](#)
 3. [Import large datasets with Python](#)
 1. [Installing Dask on your laptop](#)
 2. [Creating a Dask DataFrame](#)
 3. [Extracting information from a Dask DataFrame](#)
 4. [Importing a large dataset in Power BI with Python](#)
 4. [Importing large datasets with R](#)
 1. [Installing disk.frame on your laptop](#)
 2. [Creating a disk.frame instance](#)
 3. [Extracting information from disk.frame](#)
 4. [Importing a large dataset in Power BI with R](#)
 5. [Summary](#)
 6. [References](#)
10. [9 Calling External APIs to Enrich Your Data](#)
 1. [Technical requirements](#)
 2. [What a web service is](#)
 3. [Registering for Bing Maps Web Services](#)
 4. [Geocoding addresses using Python](#)
 1. [Using an explicit GET request](#)
 2. [Using an explicit GET request in parallel](#)
 3. [Using the Geocoder library in parallel](#)
 5. [Geocoding addresses using R](#)
 1. [Using an explicit GET request](#)
 2. [Using an explicit GET request in parallel](#)
 3. [Using the tidygeocoder package in parallel](#)
 6. [Accessing web services using Power BI](#)
 1. [Geocoding addresses in Power BI with Python](#)
 2. [Geocoding addresses in Power BI with R](#)
 7. [Summary](#)
 8. [References](#)
 11. [10 Calculating Columns Using Complex Algorithms](#)
 1. [Technical requirements](#)
 2. [The distance between two geographic locations](#)
 1. [Spherical trigonometry](#)
 2. [The law of Cosines distance](#)
 3. [The law of Haversines distance](#)
 4. [Vincenty's distance](#)

5. [What kind of distance to use and when](#)
3. [Implementing distances using Python](#)
 1. [Calculating distances with Python](#)
 2. [Calculating distances in Power BI with Python](#)
4. [Implementing distances using R](#)
 1. [Calculating distances with R](#)
 2. [Calculating distances in Power BI with R](#)
5. [The basics of linear programming](#)
 1. [Linear equations and inequalities](#)
 2. [Formulating a linear optimization problem](#)
6. [Definition of the LP problem to solve](#)
 1. [Formulating the LP problem](#)
7. [Handling optimization problems with Python](#)
 1. [Solving the LP problem in Python](#)
 2. [Solving the LP problem in Power BI with Python](#)
8. [Solving LP problems with R](#)
 1. [Solving the LP problem in R](#)
 2. [Solving the LP problem in Power BI with R](#)
9. [Summary](#)
10. [References](#)
12. [11 Adding Statistics Insights: Associations](#)
 1. [Technical requirements](#)
 2. [Exploring associations between variables](#)
 3. [Correlation between numeric variables](#)
 1. [Karl Pearson's correlation coefficient](#)
 2. [Charles Spearman's correlation coefficient](#)
 3. [Maurice Kendall's correlation coefficient](#)
 4. [Description of a real case](#)
 5. [Implementing correlation coefficients in Python](#)
 6. [Implementing correlation coefficients in R](#)
 7. [Implementing correlation coefficients in Power BI with Python and R](#)
 4. [Correlation between categorical and numeric variables](#)
 1. [Considering both variables categorical](#)
 2. [Considering a numeric variable and a categorical one](#)
 3. [Implementing correlation coefficients in Python](#)
 4. [Implementing correlation coefficients in R](#)

5. [Implementing correlation coefficients in Power BI with Python and R](#)
 5. [Summary](#)
 6. [References](#)
13. [12 Adding Statistics Insights: Outliers and Missing Values](#)
 1. [Technical requirements](#)
 2. [What outliers are and how to deal with them](#)
 1. [The causes of outliers](#)
 2. [Dealing with outliers](#)
 3. [Identifying outliers](#)
 1. [Univariate outliers](#)
 2. [Multivariate outliers](#)
 4. [Implementing outlier detection algorithms](#)
 1. [Implementing outlier detection in Python](#)
 2. [Implementing outlier detection in R](#)
 3. [Implementing outlier detection in Power BI](#)
 5. [What missing values are and how to deal with them](#)
 1. [The causes of missing values](#)
 2. [Handling missing values](#)
 6. [Diagnosing missing values in R and Python](#)
 7. [Implementing missing value imputation algorithms](#)
 1. [Removing missing values](#)
 2. [Imputing tabular data](#)
 3. [Imputing time-series data](#)
 4. [Imputing missing values in Power BI](#)
 8. [Summary](#)
 9. [References](#)
14. [13 Using Machine Learning without Premium or Embedded Capacity](#)
 1. [Technical requirements](#)
 2. [Interacting with ML in Power BI with data flows](#)
 3. [Using AutoML solutions](#)
 1. [PyCaret](#)
 2. [Azure AutoML](#)
 3. [RemixAutoML for R](#)
 4. [Embedding training code in Power Query](#)
 1. [Training and using ML models with PyCaret](#)
 2. [Using PyCaret in Power BI](#)

5. [Using trained models in Power Query](#)
 1. [Scoring observations in Power Query using a trained PyCaret model](#)
 6. [Using trained models in Script Visuals](#)
 1. [Scoring observations in a script visual using a trained PyCaret model](#)
 7. [Calling web services in Power Query](#)
 1. [Using Azure AutoML models in Power Query](#)
 2. [Using cognitive services in Power Query](#)
 8. [Summary](#)
 9. [References](#)
15. [14 Exploratory Data Analysis](#)
 1. [Technical requirements](#)
 2. [What is the goal of EDA?](#)
 1. [Understanding your data](#)
 2. [Cleaning your data](#)
 3. [Discovering associations between variables](#)
 3. [Exploratory Data Analysis with Python and R](#)
 4. [Exploratory data analysis in Power BI](#)
 1. [Dataset summary page](#)
 2. [Missing values exploration](#)
 3. [Univariate exploration](#)
 4. [Multivariate exploration](#)
 5. [Variable associations](#)
 5. [Summary](#)
 6. [References](#)
 16. [15 Advanced Visualizations](#)
 1. [Technical requirements](#)
 2. [Choosing a circular barplot](#)
 3. [Implementing a circular barplot in R](#)
 4. [Implementing a circular barplot in Power BI](#)
 5. [Summary](#)
 6. [References](#)
 17. [16 Interactive R Custom Visuals](#)
 1. [Technical requirements](#)
 2. [Why interactive R custom visuals?](#)
 3. [Adding a dash of interactivity with Plotly](#)

4. [Exploiting the interactivity provided by HTML widgets](#)
5. [Packaging it all into a Power BI Custom Visual](#)
 1. [Installing the pbviz package](#)
 2. [Developing your first R HTML custom visual](#)
6. [Importing the custom visual package into Power BI](#)
7. [Summary](#)
8. [References](#)

Extending Power BI with Python and R: Ingest, transform, enrich and visualize using the power of analytical languages

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You'll be notified when a new version is ready.

This title is in development, with more chapters still to be written, which means you have the opportunity to have your say about the content. We want to publish books that provide useful information to you and other customers, so we'll send questionnaires out to you regularly. All feedback is helpful, so please be open about your thoughts and opinions. Our editors will work their magic on the text of the book, so we'd like your input on the technical elements and your experience as a reader. We'll also provide frequent updates on how our authors have changed their chapters based on your feedback.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book. Join the exploration of new topics by contributing your ideas and see them come to life in print.

1. Where and How to Use R and Python Scripts in Power BI
2. Configuring R with Power BI
3. Configuring Python with Power BI
4. Importing Unhandled Data Objects
5. Using Regular Expressions in Power BI
6. Anonymizing and Pseudonymizing Your Data in Power BI
7. Logging Data From Power BI To External Sources
8. Loading Large Datasets Beyond the Available RAM in Power BI

9. Calling External APIs to Enrich Your Data
10. Calculating Columns Using Complex Algorithms
11. Adding Statistics Insights: Associations
12. Adding Statistics Insights: Outliers and Missing Values
13. Using Machine Learning Without Premium or Embedded Capacity
14. Exploratory Data Analysis
15. Advanced Visualizations
16. Interactive R Custom Visuals

1 Where and How to Use R and Python Scripts in Power BI

Power BI is Microsoft's flagship **self-service business intelligence** product. It consists of a set of on-premises applications and cloud-based services that help organizations integrate, transform, and analyze data from a wide variety of source systems through a user-friendly interface.

The platform is not limited to data visualization. Power BI is much more than this, when you consider that its analytics engine (**Vertipaq**) is the same as **SQL Server Analysis Services (SSAS)** and **Azure Analysis Services**. It also uses **Power Query** as its data extraction and transformation engine, which we find in both Analysis Services and **Excel**. The engine comes with a very powerful and versatile formula language (**M**) and GUI, thanks to which you can "grind" and shape any type of data into any form.

Moreover, Power BI supports **DAX** as a data analytic formula language, which can be used for advanced calculations and queries on data that has already been loaded into tabular data models.

Such a versatile and powerful tool is a godsend for anyone who needs to do data ingestion and transformation in order to build dashboards and reports to summarize a company's business.

Recently, the availability of huge amounts of data, along with the ability to scale the computational power of machines, has made the area of **advanced analytics** more appealing. So, new mathematical and statistical tools have become necessary in order to provide rich insights. Hence the integration of analytical languages such as **Python** and **R** within Power BI.

R or Python scripts can only be used within Power BI with specific features. Knowing which Power BI tools can be used to inject R or Python scripts into Power BI is key to understanding whether the problem you want to address is achievable with these analytical languages.

This chapter will cover the following topics:

- Injecting R or Python scripts into Power BI
- Using R and Python to interact with your data
- R and Python limitations on Power BI products

Technical requirements

This chapter requires you to have **Power BI Desktop** already installed on your machine (you can download it from here: <https://aka.ms/pbiSingleInstaller>).

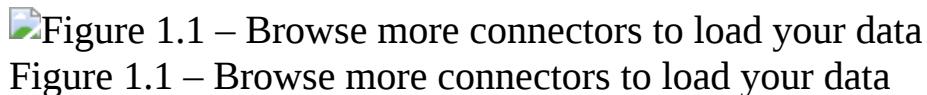
Injecting R or Python scripts into Power BI

In this first section, the Power BI Desktop tools that allow you to use Python or R scripts will be presented and described in detail. Specifically, you will see how to add your own code during the **data loading**, **data transforming**, and **data viewing** phases.

Data loading

One of the first steps required to work with data in Power BI Desktop is to **import** it from external sources:

1. There are many connectors that allow you to do this, depending on the respective data sources, but you can also do it via scripts in Python and R. In fact, if you click on the **Get data** icon in the ribbon, not only are the most commonly used connectors shown, but you can select other ones from a more complete list by clicking on **More...**:



2. In the new **Get Data** window that pops up, simply start typing the string `script` into the search text box, and immediately the two options for importing data via Python or R appear:



Figure 1.2 – Showing R script and Python script into the Get Data window

Figure 1.2 – Showing R script and Python script into the Get Data window

3. Reading the contents of the tooltip, obtained by hovering the mouse over the **Python script** option, two things should immediately jump out at you:a)A local installation of Python is required.B)What can be imported through Python is a data frame.The same two observations also apply when selecting **R script**. The only difference is that it is possible to import a **pandas DataFrame** when using Python (a DataFrame is a data structure provided by the pandas package), whereas R employs the two-dimensional array-like data structure called **R data frame**, which is provided by default by the language.
4. After clicking on the **Python script** option, a new window will be shown containing a text box for writing the Python code:



Figure 1.3 – Window showing the Python script editor

Figure 1.3 – Window showing the Python script editor

As you can see, it's definitely a very skimpy editor, but in *Chapter 3, Configuring Python with Power BI*, you'll see how you can use your favorite IDE to develop your own scripts.

5. Taking a look at the warning message, Power BI reminds us that no Python engine has been detected, so it must be installed. Clicking on the **How to install Python** link will cause a Microsoft Docs web page to open, explaining the steps to install Python. Microsoft suggests installing the base Python distribution, but in order to follow some best practices on **environments**, we will install the **Miniconda** distribution. The details of how to do this and why will be covered in Chapter 3.
6. If you had clicked on **R script** instead, a window for entering code in R, similar to the one shown in *Figure 1.4*, would have appeared:



Figure 1.4 – Window showing the R script editor

Figure 1.4 – Window showing the R script editor

As with Python, in order to run code in R, you need to install the R engine on your machine. Clicking on the **How to install R** link will open a Docs page where Microsoft suggests installing either **Microsoft R Open** or the classic **CRAN R**. *Chapter 2, Configuring R With Power BI*, will show you which engine to choose and how to configure your favorite IDE to write code in R.

In order to import data using Python or R, you need to write code in the editors shown in *Figure 1.3* and *Figure 1.4* that assigns a pandas DataFrame or an R data frame to a variable, respectively. You will see concrete examples throughout this book.

Next, let's look at transforming data.

Data transformation

It is possible to apply a transformation to data already imported or being imported, using scripts in R or Python. Should you want to test this on the fly, you can import the following CSV file directly from the web: <http://bit.ly/iris.csv>. Follow these steps:

1. Simply click on **Get data** and then **Web** to import data directly from a web page:



Figure 1.5 – Select the Web connector to import data from a web page

2. You can now enter the previously mentioned URL in the window that pops up:

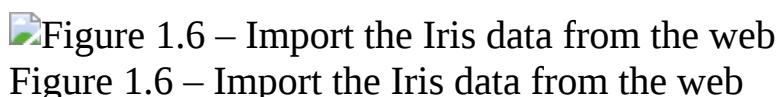


Figure 1.6 – Import the Iris data from the web

Right after clicking **OK**, a window will pop up with a preview of the data to be imported.

3. In this case, instead of importing the data as-is, click on **Transform Data** in order to access the Power Query data transformation window:



Figure 1.7 – Imported data preview

Figure 1.7 – Imported data preview

4. It is at this point that you can add a transformation step using a Python or R script by selecting the **Transform** tab in **Power Query Editor**:



Figure 1.8 – R and Python script tools into Power Query

Editor

Figure 1.8 – R and Python script tools into Power Query

Editor

5. By clicking on **Run Python script**, you'll cause a window similar to the one you've already seen in the data import phase to pop up:

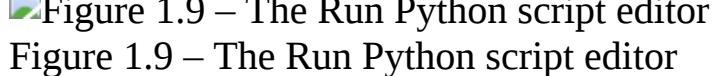


Figure 1.9 – The Run Python script editor

Figure 1.9 – The Run Python script editor

If you carefully read the comment in the text box, you will see that the dataset variable is already initialized and contains the data present at that moment in Power Query Editor, including any transformations already applied. At this point, you can insert your Python code in the text box to transform the data into the desired form.

6. A similar window will open if you click on **Run R script**:



Figure 1.10 – The Run R script editor

Figure 1.10 – The Run R script editor

Also, in this case, the dataset variable is already initialized and contains the data present at that moment in Power Query Editor. You can then add your own R code and reference the dataset variable to transform your data in the most appropriate way.

Next, let's look at visualizing data.

Data visualization

Finally, your own Python or R scripts can be added to Power BI to create new visualizations, in addition to those already present in the tool out of the box:

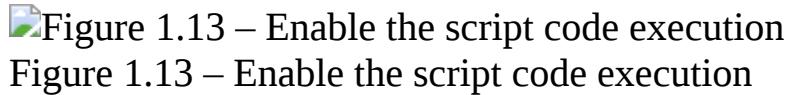
1. Assuming we resume the data import activity begun in the previous section, once the `Iris` dataset is loaded, simply click **Cancel** in the **Run R script** window, and then click **Close & Apply** in the **Home** tab of Power Query Editor:



2. After the data import is complete, you can select either the **R script visual** or **Python script visual** option in the **Visualizations** pane of Power BI:



3. If you click on **Python script visual**, a window pops up asking for permission to enable script code execution, as there may be security or privacy risks:



4. After enabling code execution, in Power BI Desktop you can see a placeholder for the Python visual image on the report canvas and a Python script editor at the bottom:



You can now write your own custom code in the Python editor and run it via the **Run script** icon highlighted in *Figure 1.14* to generate a Python visualization.

A pretty much identical layout occurs when you select **R script visual**.

Using R and Python to interact with your data

In the previous section, you saw all the ways you can interact with your data in Power BI via R or Python scripts. Beyond knowing how and where to inject your code into Power BI, it is very important to know how your code will interact with that data. It's here that we see a big difference between the effect of scripts injected via Power Query Editor and scripts used in visuals:

- **Scripts via Power Query Editor:** This type of script will transform the data and **persist** transformations in the model. This means that it will always be possible to retrieve the transformed data from any object within Power BI. Also, once the scripts have been executed and have taken effect, *they will not be re-executed unless the data is refreshed*. Therefore, it is recommended to inject code in R or Python via Power Query Editor when you intend to use the resulting insights in other visuals, or in the data model.
- **Scripts in visuals:** The scripts used within the R and Python script visuals extract particular insights from the data and only make them evident to the user through **visualization**. Like all the other visuals on a report page, the R and Python script visuals are also interconnected with the other visuals. This means that the script visuals are subject to **cross-filtering** and therefore *they are refreshed every time you interact with other visuals in the report*. That said, it is not possible to persist the results obtained from the visuals scripts in the data model.

Tip

Thanks to the interactive nature of R and Python script visuals due to cross-filtering, it is possible to inject code useful to extract **real-time insights** from data, but also from external sources

(you'll see how in *Chapter 8, Calling External APIs to Enrich Your Data*). The important thing to keep in mind is that, as previously stated, it is then only possible to visualize such information, or at the most to write it to external repositories (as you will see in *Chapter 7, Log Data from Power BI to External Repositories*).

In the final section of this chapter, let's look at the limitations of using R and Python when it comes to various Power BI products.

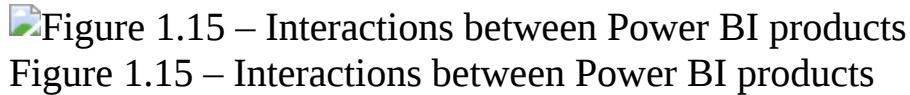
R and Python limitations on Power BI products

The first question once you are clear on where to inject R and Python scripts in Power BI could be: "*Is the use of R and Python code allowed in all Power BI products?*" In order to have a brief recap of the various Power BI products and their usage in general, here is a concise list:

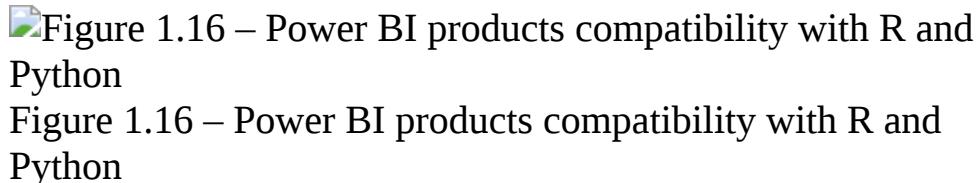
- **Power BI Service:** This is sometimes called **Power BI Online**, and it's the **Software as a Service (SaaS)** declination of Power BI. It was created to facilitate the sharing of visual analysis between users through Dashboards and Reports.
- **Power BI Report Server:** This is the on-premises version of Power BI and it extends the capabilities of **SQL Server Reporting Services**, enabling the sharing of reports created in **Power BI Desktop (for Report Server)**.
- **Power BI Embedded:** A Microsoft Azure service that allows dashboards and reports to be embedded in an application for users who do not have a Power BI account.
- **Power BI Desktop:** A free desktop application for Windows that allows you to use almost all of the features that Power BI exposes. It is not the right tool for sharing results between users, but it allows you to share them on Power BI Service and Power BI Report Server. The desktop versions that allow publishing on the two mentioned services are distinct.

- **Power BI Mobile:** A mobile application, available on Windows, Android, and iOS, that allows secure access to Power BI Service and Power BI Report Server, and that allows you to browse and share dashboards and reports, but not edit them.

Apart from the licenses, which we will not go into here, a summary figure of the relationships between the products mentioned previous follows:



Unfortunately, of all these products, only **Power BI Service**, **Power BI Embedded**, and **Power BI Desktop** allow you to enrich data via code in R and Python:



Important Note

From here on out, when we talk about *Power BI Service* in terms of compatibility with analytical languages, what we say *will also apply to Power BI Embedded*.

So, if you need to develop reports using advanced analytics through R and Python, make sure the target platform supports them.

Summary

This chapter has given a detailed overview of all the ways by which you can use R and Python scripts in Power BI Desktop. During the data ingestion and data transformation phases, Power Query Editor allows you to add steps containing R or Python code. You can also make use of these analytical languages during the data visualization phase thanks to the R and Python script visuals provided by Power BI Desktop.

It is also very important to know how the R and Python code will interact with the data already loaded or being loaded in Power BI. If you use Power Query Editor, both when loading and transforming data, the result of script processing will be persisted in the data model. Also, if you want to run the same scripts again, you have to refresh the data. On the other hand, if you use the R and Python script visuals, the code results can only be displayed and are not persisted in the data model. In this case, script execution occurs whenever cross-filtering is triggered via the other visuals in the report.

Unfortunately, at the time of writing, you cannot run R and Python scripts in any other Power BI product. The only ones that provide for running analytics scripts are Power BI Desktop and Power BI Service.

In the next chapter, we will see how best to configure the R engine and RStudio to integrate with Power BI Desktop.

2 Configuring R with Power BI

Power BI Desktop is not equipped with the analytical language engines presented in the previous chapter by default. Therefore, it is necessary to install these engines and properly configure Power BI Desktop to correctly interface with them. It is also recommended to install an **Integrated Development Environment (IDE)**, enabling you to work in the way you are most comfortable.

We'll look at how to get those engines up and running and give you some general guidelines on how to pick the most appropriate one for your needs. After that, we'll look at how to make these engines interface with both Power BI Desktop and the Power BI service.

Finally, we will give some important tips on how to overcome some stringent limitations of R visuals on the Power BI service.

In particular, this chapter will deal in detail with the following topics:

- The available R engines
- Choosing an R engine to install
- Installing an IDE for R development
- Configuring Power BI Desktop to work with R
- Configuring the Power BI service to work with R
- Limitations of R visuals

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. It also requires you to have signed up for the Power BI service in the last part of the chapter (here's a how-to: <http://bit.ly/signup-powerbiservice>). A **Power BI free** license is enough to test all the code in this book, as you will share reports only in your personal **workspace**.

The available R engines

There is more than one R distribution available on the market that you can use for free for your advanced analytics projects. In this section, we'll explore the main details of each of them.

The CRAN R distribution

When it comes to installing the R engine, we almost always think of the open source software environment *par excellence*, developed by a collective of contributors over the years, known as **CRAN R**, also called **base R** (<https://cran.r-project.org>). To be exact, the **Comprehensive R Archive Network (CRAN)** is a network of web servers and FTP servers around the world, whose goal is to preserve multiple identical and up-to-date versions of the R source code and the entire ecosystem of R packages developed by the community, along with all the R documentation.

One of the biggest advantages of CRAN R is its very active community of developers. Their contribution to the creation of new packages on CRAN is invaluable. That's why if you think you need a particular feature to process your data, it's almost certain that it has already been developed by the R community and released as a free usable R package.

However, not everyone knows that CRAN R is not the only R distribution available on the market.

The Microsoft R Open distribution and MRAN

Even Microsoft has contributed to the community by releasing its own open source R "distro" for both Windows and Linux, under the terms of the *General Public License version 2*. Starting from 2016, Microsoft has released its own distributions of R, called **Microsoft R Open (MRO)** and sometimes referred to simply as **Microsoft R**, which reflects the same versions released by CRAN R and is 100% compatible with it.

Important note

If you take any code written in CRAN R, using any CRAN package, and run it with the same version of the MRO engine, everything will work fine.

In addition, there are only a few Microsoft-owned functions (including **RevoUtils** and **RevoUtilsMath**) that return information about the engine installation (such as path and memory usage) and the maximum number of threads that the **Math Kernel Library (MKL)** can run.

The benefit of the MRO release was that it brought important improvements for the R community:

- The multi-threaded **Intel MKL** included out of the box
- A high-performance mirror of the CRAN repository, called **Microsoft R Application Network (MRAN)**, (<https://mran.microsoft.com>), that gives you a “time machine” tool to get a snapshot of CRAN packages at the selected time

Let's see in detail what they are.

Multi-threading in MRO

Although CRAN R was created as single-threaded, it has the ability to link to the multi-threaded **Basic Linear Algebra Subprograms (BLAS)** and **Linear Algebra Package (LAPACK)** libraries too, but these are not straightforward activities to do manually.

Installing MRO offers the advantage of having the engine already preconfigured and optimized for Intel processors thanks to MKL, with the purpose of performing mathematical calculations in parallel. To get an idea of the gain in computational time, take a look at the benchmark results obtained comparing R-3.4.1 and MRO 3.4.1 in *Figure 2.1* (source:<http://bit.ly/msropen-bnchmrk>):

Figure 2.1 – Total elapsed time for each of the benchmark tests on the same machine

Figure 2.1 – Total elapsed time for each of the benchmark tests on the same machine

As you can easily see, the time taken by CRAN R (installed as-is without linking the BLAS libraries) in the matrix calculation is significantly higher than that taken by MRO. It is therefore strongly recommended to use MRO if you make heavy use of math routines for data science, engineering, financial analysis, and so on.

Reproducing results with checkpoints

When performing complex analyses using R, it is very likely that the resulting script is based on one or more CRAN packages. One of the most common problems in R is the reproducibility of results, because packages on CRAN change every day. So, it can happen that suddenly a script that was working fine a few weeks ago starts to generate errors never seen before after you update the packages, or when you run it on another machine. Even more insidious is the possibility of getting incorrect results without realizing it, again due to a package update.

For this reason, starting in September 2014, Microsoft put up its own distribution of CRAN's R packages organized into daily snapshots of the entire repository, taken by the **checkpoint server** at midnight UTC and persisted on MRAN:



Figure 2.2 – The process of persisting a daily snapshot of CRAN in MRAN

Figure 2.2 – The process of persisting a daily snapshot of CRAN in MRAN

From that point on, the R community had the ability to access a specific snapshot of CRAN packages with their versions frozen to a specific date.

Important note

It is not necessary to use MRO to benefit from the free service offered by the checkpoint server. CRAN R users can also use it.

Usually, when you invoke `install.packages` to install one or more packages in R, the latest CRAN version of each specified package is installed. Using the `checkpoint` package, you have the ability to reference

a specific snapshot of the CRAN repository on a specific day in the past, as follows:

```
library(checkpoint)
checkpoint("YYYY-MM-DD") # Replace the data you want here
install.packages("dplyr")
```

With the preceding code example, both you and any other user are sure to always use the same version of the `dplyr` package downloaded from the CRAN repository snapshot you want to use.

The MRO installation includes the `checkpoint` package and by default sets the snapshot date as equal to the release date of the MRO version you are installing. You can easily verify the snapshot date that is set by the specific version of MRO in the **CRAN Time Machine** tool available on the MRAN portal (<https://mran.microsoft.com/timemachine>):



Figure 2.3 – The process of persisting a daily snapshot of CRAN in MRAN

Figure 2.3 – The process of persisting a daily snapshot of CRAN in MRAN

You can also check the default package repository programmatically in the following way:

```
getOption("repos")
```

And you'll get something like this as output:

```
CRAN
"https://mran.microsoft.com/snapshot/2020-07-16"
```

With this example code, if you install any R package, the versions installed will be those that were present in CRAN at midnight on July 16, 2020.

Microsoft R Client

Microsoft has also released another distribution of R, available for free for Windows and Linux, called **Microsoft R Client**. This distribution is

entirely based on MRO, allowing you to run any code that works on CRAN R. In addition, there are some of Microsoft's R function libraries (<http://bit.ly/ml-r-funcions>) that are not open sourced. These functions are multi-threaded and allow you to work even with datasets that normally could not be contained in the memory of the machine with Microsoft's on-premises products (**SQL Server** and **Machine Learning Server**). Yes, that's right: in case you didn't know, Microsoft's **Relational Database Management System (RDBMS)**, SQL Server, lets you use R, Python, and Java code in its stored procedures. So, if your data source is SQL Server itself and you need to extract insights from the data that are persisted there, consider also the possibility of applying Advanced Analytics algorithms directly using stored procedures in SQL Server, embedding R and Python code within them.

Using Microsoft R Client, it is possible to delegate algorithms that require non-trivial computational complexity to SQL Server or Machine Learning Server thanks to the performant **MicrosoftML** and **RevoScale** packages mentioned previously.

Unlike MRO, Microsoft R Client is not updated in parallel with each individual CRAN R release, but follows the MRO releases included at the time of Machine Learning Server releases. For example, as of today, CRAN R is at version 4.0.2, and MRO is at version 4.0.2 too, whereas Microsoft R Client is at version 3.5.2, which is the one supported by Machine Learning Server 9.4.

Having described the main features of the most popular R distributions, let's move on to the selection of the engine to be installed on your machine.

Choosing an R engine to install

The first question when installing an R engine is, *"Which distribution should I install? Do I choose the standard CRAN R, or do I opt for any Microsoft R distribution?"* The usual answer to these kinds of questions is, *"It depends!"* In our case, the goal is to use the R engine within Power BI, so we need to understand which engines are used by the different products that permit the use of R within them.

The R engines used by Power BI

We saw in *Chapter 1, Where and How to Use R and Python Scripts in Power BI*, that only two Power BI products are allowed to use scripts in R and Python: Power BI Desktop and the Power BI service (remember that Power BI Embedded is implicitly included when talking about the Power BI service). So, the answer "*It depends!*" takes on a clearer connotation now: if you need to share your reports with people inside your organization, then you have to install the engines that work well with the Power BI service; if, instead, you need to create reports for your own use, without even publishing it on **My Workspace**, you can install the engines suitable for Power BI Desktop.

There is a substantial difference in the use of analytic engines by these two products. **Power BI Desktop** relies on the R engine installed by the user on the same machine on which Power BI Desktop is running. It is the user who chooses which version of the engines and which packages to install. Power BI Desktop simply ensures that any R code entered through its interface runs directly on that engine.

The **Power BI service** is the **Software-as-a-Service (SaaS)** product among those covered by Power BI. The user doesn't have to take on the maintenance of its underlying IT infrastructure, and nor can the user decide to install components on it at will.

Important note

The R engine and the packages used by the Power BI service for **R visuals** are preinstalled on the cloud and therefore the user must adapt to the versions adopted by the service.

It is also important to know exactly which R distribution is used by the R visuals in the Power BI service.

Important note

To date, the Power BI service relies on the **MRO 3.4.4** runtime when implementing an R visual. It is important to always keep an

eye on the version of the R engine and the packages provided by the Power BI service with each release to ensure that the reports to be published work properly. See the following link for more information: <http://bit.ly/powerbi-r-limits>.

If, instead, you have to do data ingestion or data transformation using R scripts and you need to refresh your data, the Power BI service does not use the same engine for R visuals in this case.

Important note

The R engine used by the Power BI service during the data refresh phase for *R scripts in Power Query* has to be installed on any machine of your choice outside the service, and on that same machine you have to install the **on-premises data gateway** in **personal mode**. Note that you must use external engines even if the data to be refreshed does not flow through the gateway, but comes from data sources not referenced by the gateway itself.

As long as the R engine to be referenced via the data gateway is only one, it is sufficient that both are installed on the same machine. Otherwise, the following note applies.

Important note

If you need to *use multiple R engines* installed on the machine for your Power Query transformations, you must *also install Power BI Desktop*. It allows you to switch the routing of the data gateway to the selected engine through its options, updating the configuration file C:\Users\<your-username>\AppData\Local\PowerBIScripting\RSettings.xml. This file allows the override of the R engine referenced by the data gateway by default.

If you need to learn more about the on-premises data gateway (*Enterprise* and *personal* modes), we suggest reading the Microsoft Docs page at this link: <http://bit.ly/onprem-data-gateway>.

In summary, with the preceding scenarios in mind, if you need to do reports for personal use on your desktop, you have no limitations on which R engine to use, so you can install the R versions and packages that suit you best. If, however, you know in advance that the reports you are going to create contain R visuals and are intended to be shared with colleagues on the Power BI service, there are stringent limitations on both the version and the packages to be pre-installed in the service.

Let's now move on to the installation of the engines.

Installing the suggested R engines

Managing dependencies of R scripts injected within reports developed for the Power BI service can be complex in the long run. Keeping in mind that it is possible to install more than one R engine on the same machine, we suggest the following tip.

Tip

We recommend that you *dedicate a machine* to run only the R engines used by Power BI reports. Our suggestion is to install an R engine for each possible need that may arise when developing R scripts in Power Query or in R visuals.

The R engine for data transformation

You have seen that R scripts used in Power Query to make changes to the data model must necessarily use an external R engine through the on-premises data gateway in personal mode, even if you use the Power BI service.

Tip

We recommend installing the latest available version of either CRAN R or MRO. Since both distributions are interchangeable in terms of code operation given the same version, in our opinion it

is better to install *MRO* to benefit from the performance advantages of the MKL library.

Installing MRO is very simple:

1. Go to <https://mran.microsoft.com/download>.
2. Click on the **Download** button corresponding to your OS. In my case, this is Windows:
Figure 2.4 – Downloading MRO for Windows
Figure 2.4 – Downloading MRO for Windows
3. Once the `microsoft-r-open-x.x.x.exe` file (where `x.x.x` corresponds to the actual version chosen) is downloaded, double-click on it and then click on **Continue** on the **Let's get started** window that pops up.
4. The next screen, **Configure the installation**, asks you to confirm or change the installation path, as well as select some installation options:



The only optional choice is to install **MKL**. For reasons you know well by now, we suggest installing it. Then click on **Continue**.

5. In the next window, you can read the Microsoft license agreement to install MRO. After checking **I acknowledge the above licensing information**, click on **Continue**.
6. Similarly, in the next window you can read the license agreement for MKL. After ticking **I accept these terms**, click on **Continue**.
7. A summary window now appears asking you to click on **Install** to begin installing MRO.
8. At the end of the installation, an **All done** window will inform you that the installation was completed successfully.

And that's it! You are now ready to be able to write and run your R code on MRO.

Important note

Usually, the Power BI Desktop installation on which you develop reports is located on a separate machine from the one selected as the Power BI service's R engine machine, where the data gateway is also often installed. In that case, you must also install the R engine on the machine on which your Power BI Desktop instance is installed.

If you want, you can already write code on the very rudimentary GUI installed by default with MRO:



We'll see later in this chapter how to install the IDE preferred by all R developers: **RStudio**.

The R engine for R visuals on the Power BI service

As previously mentioned, R visual scripts published on the Power BI service run on a pre-installed R engine on the cloud, the version of which may change based on new releases of the Power BI service itself. Should you need to share a report containing an R visual with colleagues, you need to be sure that your R code works correctly on the pre-installed engine.

Tip

We strongly recommend that you also install on your machine the *same version* of MRO as that used for R visuals by the Power BI service.

In order to install the same version of MRO provided by the Power BI service, the process is very simple:

1. Go to <http://bit.ly/powerbi-r-limits> and check the actual version of MRO used by the Power BI service:

Figure 2.7 – The actual MRO version used by the Power BI service

Figure 2.7 – The actual MRO version used by the Power BI service

2. Then go to <https://mran.microsoft.com/release-history> and check for the same MRO version you saw in step 1 (in our case, **3.4.4**):

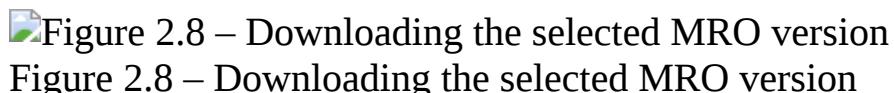
Figure 2.8 – Downloading the selected MRO version

Figure 2.8 – Downloading the selected MRO version

3. After downloading the executable, simply follow the steps outlined in the previous section to install this specific version of the MRO engine.

What to do when the Power BI service upgrades the R engine

It may happen that you have already installed a version of MRO specifically to be aligned with the R visual engine on the Power BI service at a certain date, and after a while Microsoft updates the Power BI service, also upgrading the pre-installed R engine. This scenario causes two separate events:

- The R engine is more up to date than its predecessor and may contain breaking changes that could make previously developed reports unusable. This is a very rare eventuality, since very often such changes are mitigated by the introduction of “deprecated” functions or parameters, thus ensuring backward compatibility.
- Many of the packages installed on the release date of the previous R engine will be updated to a later version, corresponding to the release date of the new version of the R engine. Updating a package to a newer version is one of the most frequent causes of errors due to the incompatibility of a script written for the previous version of the package.

In this case, you should verify that all of your reports that have R visuals and are published to the Power BI service also work with the updates made to the service.

Tip

In light of the preceding observations, and considering the fact that the pre-installed R engine of the Power BI service is very rarely updated, it is better to directly install the new version of MRO on your machine and test your reports making sure that Power BI Desktop references the new version of the engine. You need to fix any code issue in those R visuals that have some problems, after which you can publish those reports back to the Power BI service.

Once you've made sure that all of the preceding reports are working properly, it's up to you to decide whether you want to uninstall the previous version of MRO to free up disk space and to avoid confusion in handling these particular reports.

At this point we can move on to install an IDE that has more features than the rGUI installed by default by MRO.

Installing an IDE for R development

The need to install a state-of-the-art IDE for the development of code in Power BI comes from the need to have all the tools necessary to identify any bugs and to quickly test the results of code chunks on the fly.

Tip

It is strongly suggested to test your R code in the IDE and verify the results before using it in Power BI.

There are many IDEs for R development on the market. Some examples are **R-Brain IDE (RIDE)**, **IntelliJ IDEA**, and **Jupyter Lab**, but it is estimated that over 90% of R programmers use **RStudio** as their primary IDE. That's

why we suggest you also use this IDE to test the code you'll encounter throughout this book.

Installing RStudio

Installing RStudio on your machine is very simple:

1. Go to <https://rstudio.com/products/rstudio/download/> and click on **Download** under the **RStudio Desktop** column:



2. Then click on the **Download RStudio for Windows** button:



3. After the download completes successfully, double-click on the executable and click on the **Next** button on the RStudio setup welcome window.
4. In the next window, you are asked for the folder in which to install RStudio. Leave the default one and click on **Next**.
5. The following window asks you to select the Start menu folder in which to create the shortcuts. Leave the default one and click on **Install**.
6. When the installation is complete, click on **Finish**.

Just to make sure you installed everything smoothly, check that you have the option to select one of the two newly installed R engines from the RStudio options:

1. Open the newly installed RStudio from the **Start** menu:



2. Once opened, you may be asked if you want to anonymously send any crash reports to RStudio to improve the product. Select **Yes** or **No** as your choice.
3. As you can see, RStudio has already selected an R engine by default. In our case, it is the latest version installed on our machine. Also, note that MKL is up and running:



Figure 2.12 – The default R engine selected in RStudio and Global Options

Now, click on the **Tools** menu and then **Global Options...** to see the available engines.

4. In the **General** menu of the **Options** window, you have the possibility to select the engine you want to use in RStudio by clicking on **Change...** and then selecting one of the available engines:

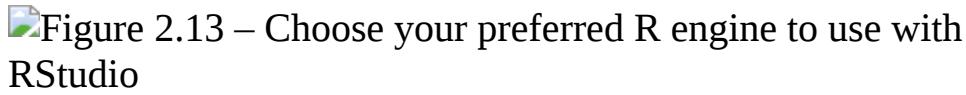


Figure 2.13 – Choose your preferred R engine to use with RStudio

Usually, we would not advise considering the engine containing "C:\PROGRA~1\...\..." as the root of the path. It is simply the engine that is currently selected in RStudio. Just select one of the other engines from the list.

5. For now, select the most recent version of R (in our case, 4.0.2) and click **OK** on the **Choose R Installation** window. You will be notified that you will need to restart RStudio for the selected changes to take effect.
6. Click **OK** on the **Change R Version** window, and then **OK** on the **Options** window. Sometimes you'll have the option to restart by selecting **Yes** in a dialog box that is shown immediately afterward. If not, just exit from RStudio and open it again.

Well done! Now you're ready to properly configure an R engine in Power BI Desktop and set up what you need to publish to the Power BI service.

Configuring Power BI Desktop to work with R

Once you have installed the R engines necessary for the development of your reports and the RStudio IDE, you must configure Power BI Desktop so that it properly references these tools. This is really a very simple task:

1. In Power BI Desktop, go to the **File** menu, click on the **Options and settings** tab, and then click on **Options**:



Figure 2.14 – Opening the Power BI Desktop Options and settings window

2. In the **Options** window, click on the **R scripting** tab on the left. The contents of the panel on the right will update, giving you the option to select the R engine to reference and the R IDE to use for R visuals:

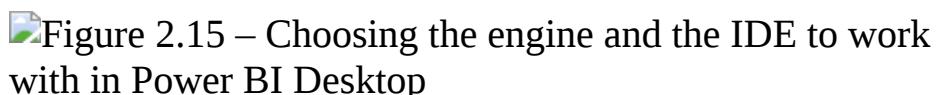


Figure 2.15 – Choosing the engine and the IDE to work with in Power BI Desktop

3. As you can see, Power BI Desktop automatically identifies the installed R engines and IDEs. For the moment, select the latest version of the engine (in our case, it is 4.0.2), in order to be aligned with the one already selected in RStudio.

You will see how to interact with the IDE from Power BI Desktop when we introduce the R and Python script visuals in *Chapter 12, Exploratory Data Analysis*.

Configuring the Power BI service to work with R

As you learned in the *The R engines used by Power BI* section of this chapter, the Power BI service uses different R engines depending on whether the scripts are used in R visuals or in Power Query for data transformation. In the first case, the engine is pre-installed on the cloud; in the second case, you need to install the **on-premises data gateway** in **personal mode** on any machine of your choice in order to make the Power BI service communicate with the R engine you installed on that machine.

Installing the on-premises data gateway in personal mode

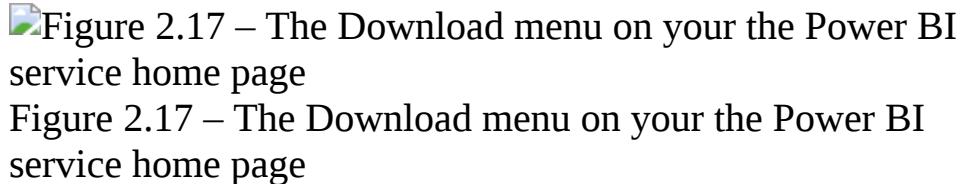
We have emphasized the fact that you will need to install the data gateway in personal mode for an important reason: R scripts are *not supported* for the on-premises data gateway in *Enterprise mode*.

In your case, you will install the data gateway on the same laptop on which you have installed the R engines and Power BI Desktop. The steps to do this are as follows:

1. Make sure you can log in to the Power BI service (<https://app.powerbi.com>). You will see a home page like the following, customized with your company logo:



2. On the top right-hand side of the home page, you'll notice a downward-pointing arrow that allows you to access the download menu. Click on this, and then select **Data Gateway**:



3. You will be redirected to a web page from which you can download the data gateway. Be sure to download the **personal mode** version:

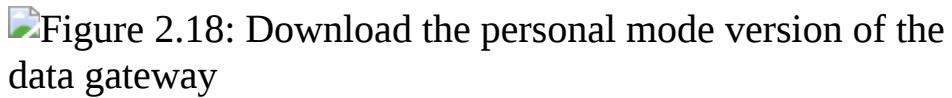


Figure 2.18: Download the personal mode version of the data gateway

4. Running the newly downloaded executable immediately opens an initial window that asks you to check the minimum installation requirements given in a link, set the installation folder of the software, and accept the terms of use for the software:



Figure 2.19 – The data gateway installation window

You can leave the default folder selected, click on the terms of use acceptance check box, and click **Install** to move on.

5. Once the data gateway installation is complete, a sign-in window will open where you will need to enter the email address with which you registered with the Power BI service, and then your password:



Figure 2.20 – The data gateway sign-in window

6. When the sign-in operation is successful, a green tick will appear with the words **The gateway is online and ready to be used**:



Figure 2.21 – The data gateway up and running

7. You can close the on-premises data gateway window.

At this moment, the Power BI service is directly connected to your machine and can access it whenever there is a need to refresh a published dataset with which your newly installed data gateway is associated.

Important note

Keep in mind that a Power BI user can *only have one* data gateway associated. If you try to install multiple on-premises data gateways in personal mode on several different machines, logging in on one of them will force a disconnect on all other data gateways.

Sharing reports that use R scripts in the Power BI service

In the previous sections you learned that, thanks to the data gateway newly installed in personal mode, you can publish a report that uses R scripts in the Power BI service to your personal workspace for sure. But, assuming you have a Pro license, can you also publish this report to shared workspaces? There is often a lot of confusion about this.

Important note

No one forbids you from publishing your report on shared workspaces as well! Although you have used a data gateway in personal mode, other users can view your report. But not only that! Other users can also refresh the dataset behind the report. They will do this "on your behalf," pointing to the machine referenced by your data gateway. The important thing is that your machine is turned on!

But then, if you can still share your reports with others using a data gateway in personal mode, what is the advantage of using a data gateway in Enterprise mode?

Using *just one* data gateway in Enterprise mode, *multiple users* in your organization can access on-premises data to which they already have access permission. It goes without saying that other users can view and refresh your report, but they cannot develop their own reports using R code by referencing your machine with R engines through your personal data gateway. This one is the true limitation.

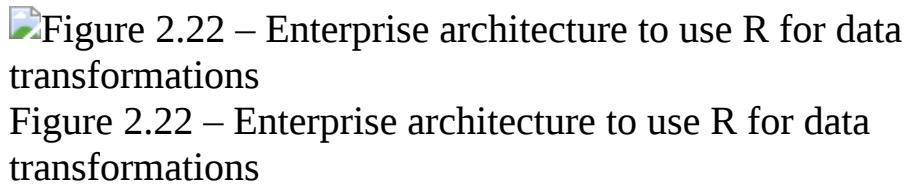
In light of the preceding, there is an **unofficial architecture**, frequently used in companies, that allows *all reports* that make use of R code in Power

Query to reference a *single machine* on which the R engine has been installed using a personal data gateway.

Tip

This unofficial architecture makes use of a personal data gateway associated not with a physical person, but with a *fictitious "service" user*. The credentials of this user are shared between all those analysts who use R code to transform data into their reports. Also, since the machine with the R engine referenced by the data gateway is shared, it must remain on during periods of scheduled activity. For this reason, an Azure Windows Virtual Machine on which both the R engine and the data gateway run is often used in this architecture.

Since a picture is worth a thousand words, *Figure 2.22* summarizes this architecture:



Thanks to this architecture it is possible to allow a group of analysts to use R scripts in their reports, despite the limitations imposed by the on-premises data gateway in personal mode.

That said, in addition to the limitations seen for R scripts in Power Query, there are some important ones to be aware of for R visuals as well.

R visuals limitations

R visuals have some important limitations regarding the data they can handle, both as input and output:

- An R visual can handle a *dataframe with only 150,000 rows*. If there are more than 150,000 rows, only the first 150,000 rows are used and

a relevant message is displayed on the image.

- R visuals have an *output size limit of 2MB*.

You must also be careful not to exceed the *5 minutes of runtime calculation* for an R visual in order to avoid a time-out error. Moreover, in order not to run into performance problems, note that *the resolution of the R visual plots is fixed at 72 DPI*.

As you can imagine, some limitations of R visuals are different depending on whether you run the visual on Power BI Desktop or the Power BI service.

To create reports in *Power BI Desktop*, you can do any of the following:

- Install any kind of package (CRAN, GitHub, or custom) in your engine for R visuals.
- Install only CRAN packages in your engine for custom R visuals (you will see an example in *Chapter 14, Interactive Custom Visuals in R*).
- Access the internet from both an R visual and a custom R visual.

When creating reports in *the Power BI service*, note the following:

- For both R visuals and custom R visuals, you can use *only the CRAN packages listed at this link: <https://bit.ly/powerbi-r-limits>*.
- You cannot access the internet from either R visuals or custom R visuals.

As you may have noticed, although the CRAN packages that can be used in an R visual on the Power BI service are limited, you can still count more than 900 packages in that list! In addition to the fact that R was the first analytical language introduced into Power BI, this shows that R is de facto considered one of the most versatile and professional languages for creating engaging visualizations.

As you probably already know, once one of your reports is published on the Power BI service, you can decide to share it on your blog, on one of your websites, or on social media via the **Publish to web** option. R visuals are not allowed to be published to the web.

Important note

Custom R visual overcomes the limitation on publishing to the web.

So, if you have to publish a report to the web, and that report contains a visualization created using R code, you must necessarily develop a custom R visual.

Summary

In this chapter, you learned about the most popular free R engines in the community. In particular, you learned what advantages have been introduced by Microsoft releasing its distribution of R to the market.

Taking note of the unique features of Power BI Desktop and the Power BI service, you have learned how to properly choose the engines and how to install them.

You have also learned about the most popular IDE in the R community and how to install it.

In addition, you were introduced to all of the best practices for properly configuring both Power BI Desktop and the Power BI service with R, whether in a development or enterprise environment.

Finally, you've learned some of the limitations on using R with Power BI, knowledge of which is critical to avoid making mistakes in developing and deploying reports.

In the next chapter, we'll see which Python engines and IDEs to install and configure in Power BI in order to develop and test Python scripts in comfort.

3 Configuring Python with Power BI

Just as in *Chapter 2, Configuring R with Python*, you had to install the R engines in order to interact with Power BI, in the same way you will also have to install the Python engines on your machine. You'll also see how to configure some IDEs so you can develop and test Python code comfortably before using it in Power BI. Therefore, similar to what we have already seen in *Chapter 2, Configuring R with Python*, the following topics will be discussed in this chapter:

- The available Python engines
- Which Python engine should I install?
- Installing an IDE for Python development
- Configuring Power BI Desktop to work with Python
- Configuring the Power BI service to work with Python
- Limitations of Python visuals

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. The last part of the chapter also requires you to be signed up for the Power BI service (here's a how-to: <http://bit.ly/signup-powerbiservice>). A **Power BI free** license is enough to test all the code in this book, as you will share reports only in your personal **workspace**.

The available Python engines

As with R, there are several distributions you can install for Python: **standard Python**, **ActivePython**, **Anaconda**, and so on. Typically, “pure” developers download the latest version of the Python engine from <https://www.python.org/>, and then install various community-developed packages useful for their projects from the **Python Package Index (PyPI)**. Other vendors, such as ActiveState and Anaconda, pre-package a specific version of the Python engine with a set of packages for the purpose of accelerating a project's startup. While the standard Python and ActiveState distributions are more aimed at general-purpose developers, Anaconda is the distribution preferred by data scientists and by those who work more closely with machine learning projects. In turn, Anaconda comes in two distinct distributions itself: Anaconda and **Miniconda**.

The Anaconda distribution, with its more than 150 included packages, can be considered to be the best do-it-yourself supermarket for data scientists, where everything is ready and configured to be used. The Miniconda distribution, on the other hand, is considered the

minimum indispensable toolbox for the data scientist seeking to trim the resources to the right level.

But there is one fundamental tool that Anaconda and Miniconda have in common: it is **Conda**, one of the most popular package managers for Python. Conda provides the developer with an easy-to-use system for the management of so-called **virtual environments**. A virtual environment, or **environment** for short, aims to encapsulate the installation of a Python engine with a set of version-specific packages. The goal is to create an isolated environment, often associated with a project or task, that can guarantee the **reproducibility of results**. This is a very important concept, essential to ensure that Python projects run smoothly when dealing with a large community of developers who create and maintain their own packages independently of each other.

Note

Contrary to what you saw in *Chapter 2, Configuring R with Power BI*, the Python community does not have a "time machine" available that easily references a specific version of the Python engine at its release date and a snapshot of the entire ecosystem of PyPI packages at the versions they were at on that date. It is up to you to build your own "time capsules" using environments in order to ensure the reproducibility of your code.

Conda is a very versatile tool. Besides managing the environments, it can also install various packages (regardless of the programming language used, not only Python), carefully managing all their dependencies. But the official recommended tool for installing Python packages from PyPI is **pip**, which only installs packages written in Python and is generally installed together with the Python engine.

That said, beyond the extent of the "bodywork" mounted around the Python engine, the various Python distributions do not add features that dramatically improve engine performance, unlike what we saw in *Chapter 2, Configuring R with Power BI*, with Microsoft R engines. For this reason, we won't go into detail about the features installed by each distribution.

Choosing a Python engine to install

Back to our scenario, to develop Python code for use in Power Query or Python visuals, what you need for sure is the following:

- A Python engine
- A package manager, to install the minimum number of packages needed to transform the data or visualize it appropriately

To select the products that best suit your needs, you will need to understand your Power BI requirements in more detail.

The Python engines used by Power BI

Just as with R visuals in the Power BI service, the following note applies to Python visuals.

Important note

The Python engine and packages used by the Power BI service for **Python visuals** are preinstalled on the cloud and therefore the user must adapt to the versions adopted by the service.

As you can imagine, the version of the engine adopted by the Power BI service is a bit behind the latest release (now 3.9.1). See the following note for more details.

Important note

To date, the Power BI service relies on the **Python 3.7.7** runtime when implementing a Python visual. It is important to always keep an eye on the version of the Python engine and packages provided by the Power BI service with each release to ensure that the reports to be published work properly. See the following link for more information: <http://bit.ly/powerbi-python-limits>.

The behavior of the Power BI service is the same as that we've already seen for the R script in the case of doing data transformation in Power Query.

Important note

The Python engine used by the Power BI service during the data refresh phase for *Python scripts in Power Query* has to be installed on any machine of your choice outside the service, and on that same machine you have to install the **on-premises data gateway in personal mode**. Note that you must use external engines even if the data to be refreshed does not flow through the gateway, but comes from data sources not referenced by the gateway itself.

As long as the Python environment to be referenced via the data gateway is the base one, it is sufficient that both are installed on the same machine. Otherwise, the following note applies.

Important note

If you need to *use multiple environments* installed on the machine for your Power Query transformations, you must *also install Power BI Desktop*. It allows you to switch the routing of the data gateway to the selected environment through its options, updating the configuration file at `C:\Users\<your-username>\AppData\Local\PowerBIScripting\PythonSettings.xml`. This file allows the overriding of the Python environment referenced by the data gateway by default.

In a nutshell, regardless of whether you want to run R or Python scripts, the infrastructure required by Power BI Desktop and the Power BI service is managed in the same way. Therefore, again, if you need to do reports *for personal use on your desktop*, you have no limitations on which Python engine to use, so you can install the Python versions and packages that suit you best. If, on the other hand, you know in advance that the reports you will create *contain Python visuals and are intended to be shared with colleagues* on the Power BI service, then there are strict limitations on both the version and the packages pre-installed in the service.

But let's get down to business and start installing the Python stuff!

Installing the suggested Python engines

Managing dependencies of Python scripts injected inside reports can be complex in the long run. Keeping in mind that it is possible to create multiple environments on the same machine, we suggest the following tip.

Tip

We recommend that you *dedicate a machine* to run only the Python engines used by Power BI reports. Our suggestion is to create a Python environment for each possible need that may arise when developing Python scripts in Power Query or for Python visuals. If you have already prepared a machine dedicated to running R scripts, as seen in *Chapter 2, Configuring R with Power BI*, then you could use the same machine to install Python engines on as well. Keep in mind that in this case, you need to make sure that the resources of the machine are sufficient to run all the engines and to satisfy the various requests coming from the various reports.

Let's first install the latest version of the Python engine, to be used for data transformation.

The Python engine for data transformation

Surely, to enrich your reports using Python, you won't need 150 pre-installed packages. Also, in order to easily manage your environments, Conda is a tool to include in your arsenal. Bearing in mind that the engine we are about to install will be used as an external Python engine by the Power BI service to transform data via Power Query through the on-premises data gateway in personal mode, the following tip applies.

Tip

We suggest adopting the latest version of **Miniconda** as the default distribution. This is because, besides pre-installing very few packages giving you the possibility to choose which packages to install, it also includes Conda in the distribution.

The installation of Miniconda is very simple:

1. Go to <https://docs.conda.io/en/latest/miniconda.html>.
2. Click on the latest version available for your OS (3.8 as of the time of writing):

Windows installers

Python version	Name	Size
Python 3.8	Miniconda3 Windows 64-bit	57.0 MiB
	Miniconda3 Windows 32-bit	54.2 MiB
Python 2.7	Miniconda2 Windows 64-bit	54.1 MiB
	Miniconda2 Windows 32-bit	47.7 MiB

Figure 3.1 – Download the latest version available of Miniconda

3. Once the file is downloaded, double-click on it, click **Next** on the welcome windows that pops up, and then click on **I Agree** to accept the License Agreement.
4. In the next window you'll be asked if you want to install Miniconda just for you or for other users as well. Leave the default setting (only for you) and click **Next**.
5. Leave the default folder for the installation on the next screen and click **Next**. Keep in mind that the installation route is in this form:
`C:\Users\<your -username>\miniconda3`.
6. In the next window, check **Register Miniconda3 as my default Python 3.8** and click **Install**:

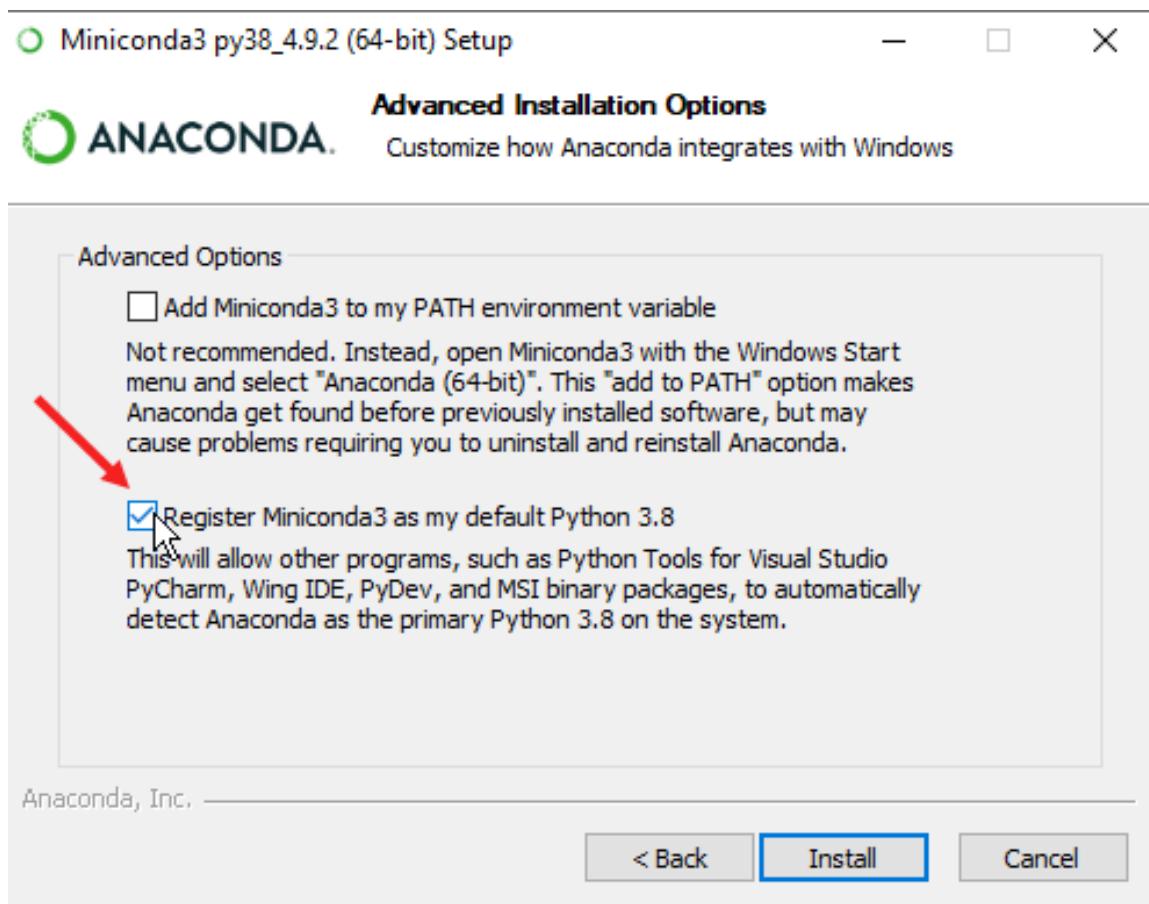


Figure 3.2 – Set Miniconda as your default Python 3.8 engine

7. At the end of the installation, an **Installation Complete** window will inform you that the installation was completed successfully. Then, click **Next**.
8. The last screen gives you the possibility to open documents containing tips and resources to start working with Miniconda. You can unflag the two options and click **Finish**.

And that's it! Now you are ready to write and run your Python code with Miniconda.

Important note

Usually, the Power BI Desktop installation on which you develop reports is located on a separate machine from the one selected as the Power BI service Python engine machine, where the data gateway is also often installed. In that case, you must also install Miniconda on the machine on which your Power BI Desktop is installed.

At the end of the installation, under the **Anaconda3 (64-bit)** folder in the Start menu, you will find shortcuts to two command-line interfaces (the standard **Command Prompt** and **PowerShell**), which ensure that you can activate **Conda** behind the scenes and interact with the tools provided by Miniconda:

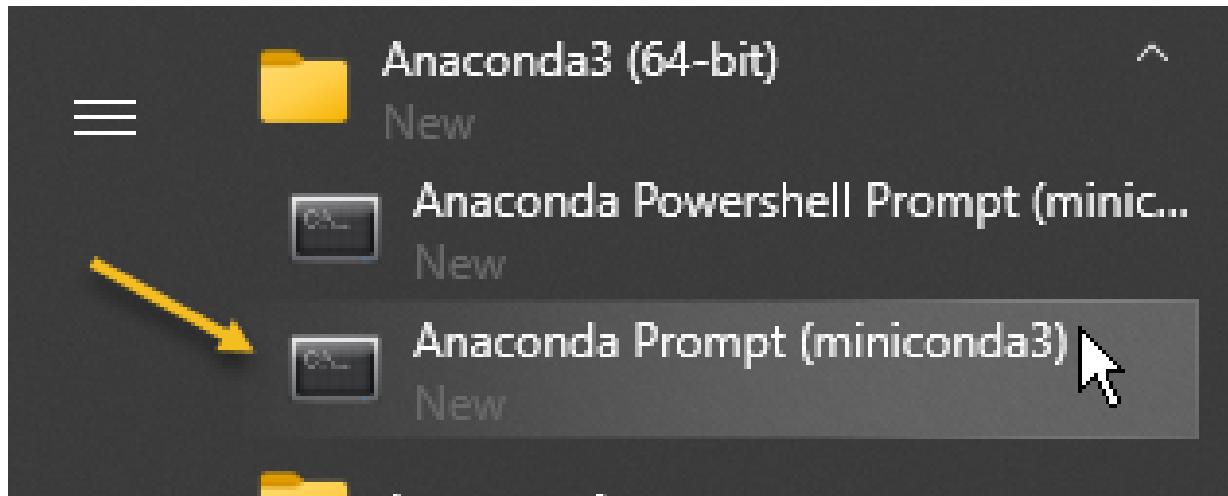


Figure 3.3 – Anaconda prompts that are useful for interacting with Miniconda

Our favorite command line is the **Anaconda Prompt** and we'll show you how to use it very shortly.

As we said in the *The available Python engines* section, both **conda** and **pip** are very good package managers. As a package dependency solver, conda is better, although a bit slower than pip. But the reason pip is often used as a package manager is that it pulls packages directly from PyPI, which is a far more complete repository than Anaconda's one. For the same reason, *we will also use pip as our default package manager*.

Creating an environment for data transformations

Contrary to what you have seen for R engines, for which two separate installations of two engines with different versions have been done, in the case of Python the installation is unique and *only the environments vary*.

Here, we will create an environment dedicated to data transformations and containing the latest version of Python made available by Miniconda and a small number of packages essential to make the first transformations.

First of all, you have to find the most recent version of Python present in the distribution you just installed:

1. Open Anaconda Prompt from the Start menu as shown previously.
2. If the prompt has small fonts, just right click on its title bar, select **Options** and then change the fonts as you like in the **Font** tab. The first thing to notice is the word **(base)** before the current path. The string before the path indicates *the name of the current environment*. The **base** environment is the default environment created during the installation of the Miniconda distribution.
3. Enter the `conda search python` command and press *Enter*.

4. You will see the list of available Python versions:

```
(base) C:\Users\LZavarella>conda search python
Loading channels: done
# Name          Version      Build Channel
python          2.7.13      h1b6d89f_16 pkgs/main
python          2.7.13      h9912b81_15 pkgs/main
python          2.7.13      hb034564_12 pkgs/main
python          2.7.14      h2765ee6_18 pkgs/main
python          3.8.5       h5rd99cc_1  pkgs/main
python          3.8.5       he1778fa_0  pkgs/main
python          3.9.0       h6244533_2 pkgs/main
python          3.9.0       h8aef87e_1 pkgs/main
python          3.9.1       h6244533_2 pkgs/main

(base) C:\Users\LZavarella>
```

Figure 3.4 – List of all the available Python versions

The latest version available in our case is **3.9.1**.

Once we have found the latest version of Python available, we can create our environment dedicated to the data transformation in Power Query, which we will call `pbi_powerquery_env`:

1. Enter the following command to create a new environment named `pbi_powerquery_env` and containing Python version `3.9.1`:

```
conda create --name pbi_powerquery_env python==3.9.1
```

You would have achieved the same thing if, instead of `==3.9.1`, you had used the form `=3.9` (with a single `=`), leaving it up to conda to find the latest micro-version.

2. Anaconda Prompt will ask you to install some packages needed to create the environment. At the `Proceed ([y]/n)?` prompt, type `y` and press *Enter*.

When the package installation is complete, you will still see **(base)** as the prompt prefix:

```
#  
# To activate this environment, use  
#  
#     $ conda activate pbi_powerquery_env  
#  
# To deactivate an active environment, use  
#  
#     $ conda deactivate
```

(base) C:\Users\LZavarella>



Figure 3.5 – After creating the new environment, you are still in the old one called “base”

This means that you are still in the base environment. Are you sure you created the new environment correctly? Let's check it:

1. Try to list the environments present on the system by entering the `conda env list` command:

```
(base) C:\Users\LZavarella>conda env list  
# conda environments:  
#  
base * C:\Users\LZavarella\miniconda3  
pbi_powerquery_env ↑ C:\Users\LZavarella\miniconda3\envs\pbi_powerquery_env
```

Figure 3.6 – List of conda environments in the system

Fortunately, the new environment is listed, but it is not the active one. The active environment is identified by an asterisk.

2. In order to install our packages inside the newly created environment, you must first **activate** it using the `conda activate pbi_powerquery_env` command:

```
(base) C:\Users\LZavarella>conda activate pbi_powerquery_env  
(pbi_powerquery_env) C:\Users\LZavarella>
```

Figure 3.7 – Activating the new environment

Now your prompt prefix correctly indicates that you are in your new environment.

3. To be on the safe side, check that the version of Python within the new environment is the one you expect with the `python --version` command:

```
(pbi_powerquery_env) C:\Users\LZavarella>python --version  
Python 3.9.1  
(pbi_powerquery_env) C:\Users\LZavarella>
```

Figure 3.8 – Checking the Python version installed in the new environment

You are inside your new environment and Python is correctly installed! You can now start installing some of the packages you'll need later. The packages to be installed are as follows:

- **NumPy**: The most widely used library in Python for working with arrays, and with functions on linear algebra, Fourier transforms, and matrices.
- **SciPy**: Used to solve scientific and mathematical problems; it is built on the NumPy extension and allows the user to manipulate and visualize data.
- **Pandas**: A Python package that provides fast, flexible, and expressive tabular, multidimensional, and time-series data.
- **Requests**: Allows you to send HTTP requests extremely easily.
- **BeautifulSoup**: A library that makes it easy to scrape information from web pages.
- **PyYAML**: Allows you to easily read and write YAML files.

You'll use the last three packages from the preceding list in the next section, where you will implicitly use web scraping procedures!

But let's get back to it, and proceed with the installation of each package via `pip`:

1. Enter the following command to install **NumPy**: `pip install numpy`.
2. Enter the following command to install **SciPy**: `pip install scipy`.
3. Enter the following command to install **Pandas**: `pip install pandas`.
4. Enter the following command to install **Requests**: `pip install requests`.
5. Enter the following command to install **BeautifulSoup**:
`pip install beautifulsoup4`.

6. Enter the following command to install PyYAML: `pip install pyyaml`.
7. Check that all packages have been installed correctly with the `conda list` command:

```
(pbi_powerquery_env) C:\Users\LZavarella>conda list
# packages in environment at C:\Users\LZavarella\miniconda3\envs\pbi_powerquery_
env:
#
# Name          Version           Build  Channel
beautifulsoup4    4.9.3            pypi_0  pypi
ca-certificates   2021.1.19        haa95532_0
certifi          2020.12.5        py39haa95532_0
chardet          4.0.0            pypi_0  pypi
idna              2.10             pypi_0  pypi
numpy             1.20.0           pypi_0  pypi
openssl          1.1.1i           h2bbff1b_0
pandas            1.2.1            pypi_0  pypi
pip               20.3.3          py39haa95532_0
python            3.9.1            h6244533_2
python-dateutil   2.8.1            pypi_0  pypi
pytz              2021.1           pypi_0  pypi
pyyaml            5.4.1            pypi_0  pypi
requests          2.25.1           pypi_0  pypi
scipy              1.6.0            pypi_0  pypi
setuptools         52.0.0          py39haa95532_0
six                1.15.0           pypi_0  pypi
soupsieve          2.1              pypi_0  pypi
sqlite             3.33.0           h2a8f88b_0
tzdata             2020f            h52ac0ba_0
urllib3            1.26.3           pypi_0  pypi
vc                 14.2              h21ff451_1
vs2015_runtime     14.27.29016       h5e58377_2
wheel              0.36.2           pyhd3eb1b0_0
wincertstore       0.2              py39h2bbff1b_0
zlib               1.2.11           h62dc9d7_4

(pbi_powerquery_env) C:\Users\LZavarella>
```

Figure 3.9 – Checking all the selected Python packages are installed

Awesome! Your new environment is now properly configured. Let's now configure another environment for Python visuals on the Power BI service.

Creating an environment for Python visuals on the Power BI service

As already mentioned, Python visual scripts published on the Power BI service run on a pre-installed Python engine on the cloud, the version of which may change with new releases of the Power BI service itself. Should you need to share a report containing a Python visual with colleagues, you need to be sure that your Python code works correctly on the pre-installed engine.

Tip

We strongly recommend that you also install on your machine the *same version* of Python that is used for Python visuals by the Power BI service.

Keep in mind that these limitations would not be there if your reports using Python visuals were not to be shared and that you only used them through Power BI Desktop. In this case, it is the engine on your machine that is used by the visuals.

To create the new environment, you must check which versions of both Python and the allowed packages are supported by the Power BI service. You can check these requirements at this link: <http://bit.ly/powerbi-python-limits>. As you can see, to date the supported version of Python is 3.7.7:

Requirements and Limitations of Python packages

There are a handful of requirements and limitations for Python packages:

- Current Python runtime: Python 3.7.7.

Figure 3.10 – Python version supported for visuals on the Power BI service

In addition, to date the only packages allowed are the following ones:

- matplotlib 3.2.1
- numpy 1.18.4
- pandas 1.0.1
- scikit-learn 0.23.0
- scipy 1.4.1
- seaborn 0.10.1
- statsmodels 0.11.1
- xgboost 1.1.0

The first thing that jumps out at you is the far smaller number of packages compared to the R packages that the Power BI service provides (8 Python packages versus more than 900 R packages!). This evident imbalance of available packages is primarily due to two causes:

- **Python** was *introduced more recently* than R (February 2019), so the Python packages introduced are mostly those essential to transforming and visualizing data.
- **R** is a language *primarily for data analysis*, and this is immediately clear because it provides a variety of packages that are designed for scientific visualization. **Python**, on the other hand, *is a general programming language* that can also be used for data analysis.

Tip

Because of the small number of Python packages supported by the Power BI service, we suggest creating a dedicated environment for Python scripts to run on the Power BI service, directly *installing all the current allowed packages*.

Keep in mind that you can't properly run a Python visual without installing some default packages. See the following note.

Important note

In order to properly run a Python visual, regardless of whether you do it on Power BI Desktop or the Power BI service, you must necessarily install the **pandas** and **Matplotlib** packages.

That said, you could already proceed to create another environment, satisfying the aforementioned version specifications, and following the steps already used to create the previous environment. Even though the Power BI service engines are updated infrequently, this manual task would still be tedious. Unfortunately, there are no ready-made "snapshots" that you can install on-the-fly to reproduce the environment, as you have seen in the case of R engines.

Tip

To avoid unnecessary manual work, we created a Python script that *scrapes the web page containing the Python engine requirements on the Power BI service* and automatically generates a **YAML file** to be used in the creation of the new environment.

YAML (defined by some funny guy using the recursive acronym **YAML Ain't Markup Language**) is a language useful for serializing data (it's a bit of a rival to JSON) and is human-readable. It is often used to serialize the contents of a computer system's configuration files.

In our case, a YAML file helps us gather together all the specifications we need to create our new environment. We thought about a YAML file because `conda` also permits the use of a YAML file as a parameter to create an environment. Our new environment, which we will call `pbi_visuals_env`, should have the following:

- The Python engine version 3.7.7
- The pip package manager
- All the 8 packages seen previously, each at the required version, installed using pip

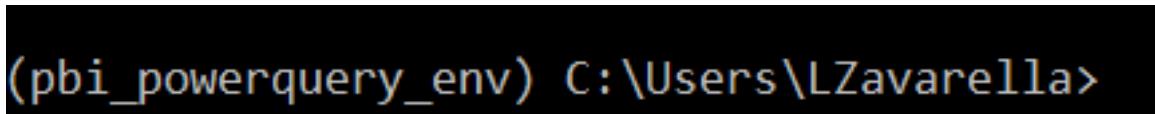
The preceding requirements can be summarized in a YAML file as follows:

```
name: pbi_visuals_env
dependencies:
  - python==3.7.7
  - pip
```

```
- pip:  
- matplotlib==3.2.1  
- numpy==1.18.4  
- pandas==1.0.1  
- scikit-learn==0.23.0  
- scipy==1.4.1  
- seaborn==0.10.1  
- statsmodels==0.11.1  
- xgboost==1.1.0
```

So, let's generate the YAML file using our Python script as follows:

1. Open your **Anaconda Prompt** (if you didn't close it before, it should still be open) and make sure that `pbi_powerquery_env` is the activated environment. If not, activate it using the `conda activate pbi_powerquery_env` command.
2. Your current path should be in the form of `C:/Users/<your-username>`. In my case, I have the following:



(`pbi_powerquery_env`) `C:\Users\LZavarella>`

Figure 3.11 – Our default Anaconda Prompt path

If not, go to your user name folder using the command:

```
cd C:/Users/<your-username> .
```

3. Let's create a new folder called `py-environments` (this will contain the Python script for web scraping, along with the YAML file) using the `md py-environments` command.
4. Now, go to the book's GitHub repository at
<https://github.com/PacktPublishing/Extending-Power-BI-with-Python-and-R>.
5. If you have already downloaded the `.zip` file of the whole repository and unzipped it on your hard drive, you will find the Python script file that we are interested in in the `Chapter03` folder, with the name
`01-create-pbi-service-py-packages-env-yaml-file.py`.
6. If you haven't downloaded the entire repository, click **Code** on top right of the page at the preceding link, and then click **Download ZIP**:

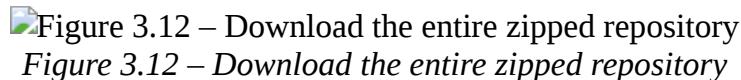


Figure 3.12 – Download the entire zipped repository
Figure 3.12 – Download the entire zipped repository

After unzipping it, you'll find the file we need in the `Chapter03` folder.

7. Now copy the file, `01-create-pbi-service-py-packages-env-yaml-file.py`, into the `C:/Users/<your-username>/py-environments` folder using the File Explorer.
8. Go back to Anaconda Prompt and change your current folder to `py-environment` using this command: `cd py-environment`.
9. Now you can finally run the Python script that does the web scraping and generates the YAML file with this command:
`python 01-create-pbi-service-py-packages-env-yaml-file.py`. You will see the contents of the YAML file printed at the prompt as follows:

Figure 3.13 – Execute the Python script to create the YAML file

Figure 3.13 – Execute the Python script to create the YAML file

You can also verify that the YAML file was generated correctly by looking again at the contents of the folder at `C:/Users/<your-username>/py-environments`:

Figure 3.14 – The YAML file correctly created

Figure 3.14 – The YAML file correctly created

10. At this point, we can directly create the new environment using the following command: `conda env create -f visuals_environment.yaml`.
11. When both Python and the packages' installations are complete, activate the newly created environment using this command: `conda activate pbi_visuals_env`.
12. Then check if the Python version is the one defined for this environment in the YAML file by entering the `python --version` command. You should see the following output:

```
(pbi_visuals_env) C:\Users\LZavarella\py-environments>python --version  
Python 3.7.7
```

Figure 3.15 – The new environment contains the right Python version

Excellent! You have finally created the new environment that will come in use when developing your Python visuals for publishing on the Power BI service. If you are interested in understanding in detail how the previous Python script managed to capture all the information needed to create the environment, you can open it in any code editor and read the very detailed comments on the code.

What to do when the Power BI service upgrades the Python engine

As we did in *Chapter 2, Configuring R with Power BI*, let's assume that you have already developed and published reports containing Python visuals using the new environment you

created earlier. Suppose that Microsoft decides to upgrade the Python version supported by the Power BI service, and consequently to upgrade the versions of currently supported packages as well. As you may have already guessed, it is likely that these updates can cause the code to fail (it is a rare event as very often, backward compatibility is guaranteed).

Tip

In such circumstances, it is often more convenient to *create a new environment on the fly*, aligned to the updated requirements from Microsoft, through the Python script you have already used previously. Next, you'll need to *test reports on the service that contain Python visuals on Power BI Desktop*, making sure that it references the newly created environment. You need to fix any code issue in those Python visuals that have some problems, after which you can publish those reports back to the Power BI service.

Once you've made sure that all of the reports are working properly, it's up to you to decide if you want to uninstall the "old" environment to free up disk space and avoid confusion in handling these particular reports.

At this point, we can move on to the configuration and installation of some IDEs that facilitate the development of Python scripts.

Installing an IDE for Python development

In *Chapter 2, Configuring R with Power BI*, you installed RStudio to conveniently develop your own R scripts. Did you know that, starting with version 1.4, you can write and run Python code directly in RStudio, making use of advanced tools for viewing instantiated Python objects?

Let's see how to configure your RStudio installation to also run Python code.

Configuring Python with RStudio

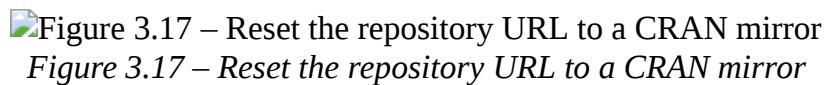
In order to allow RStudio to communicate with the Python world, you need to install a package called **reticulate**, which contains a comprehensive set of tools for interoperability between Python and R thanks to embedded Python sessions within R sessions. After that, it's a breeze to configure Python within RStudio. Let's see how to do it:

1. Open RStudio and make sure the referenced engine is the latest one, in our case **MRO 4.0.2**. As seen in *Chapter 2, Configuring R with Power BI*, you can set up your R engine in RStudio by going to the **Tools** menu and then **Global Options**....
2. Let's check at which date the current snapshot is frozen by typing `getOption("repos")` in the command-line interface:



As you can see, the snapshot date is not very recent.

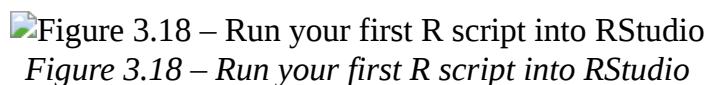
3. Since the goal of installing the latest R engine is to develop code using the latest packages, we can remove the snapshot limitation by overwriting the package repository URL. You can do this by creating a new R script, by clicking on the top-left green + icon and then choosing **R Script**:



4. Then copy the following script into the new script tab that has been added by RStudio:

```
local({  
  r <- getOption("repos")  
  r["CRAN"] <- https://cloud.r-project.org/  
  options(repos = r)  
})
```

Now, just highlight the script and click **Run** at the top right:

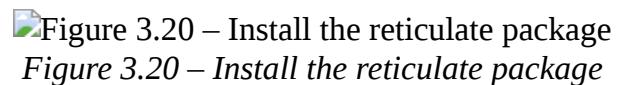


5. If you now go to the console at the bottom and enter `getOption("repos")` again, you will see the updated repository URL:



Now, you are sure to always install the latest versions of the packages on CRAN.

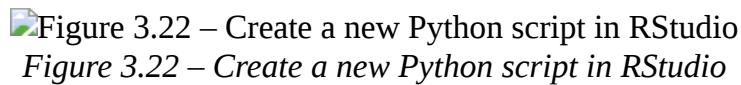
6. You can now install the reticulate package by clicking on the **Packages** tab on the bottom-right panel in RStudio, clicking on **Install**, entering the `reticulate` string into the textbox, and finally clicking on the **Install** button:



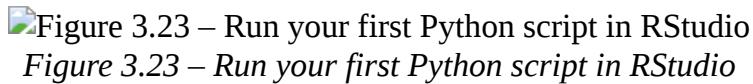
7. After that, go to the **Tools** menu and then **Global Options....** In the **Options** windows that pops up, click on **Python**, then click on **Select...** to choose your Python interpreter executable. Note that you will have as many executables as the number of environments you have created:



8. Choose the Python interpreter with the latest version (in our case 3.9.1). You'll be asked to restart the current R session. Choose **Yes**.
9. Now you can open a new Python script file by clicking on the green + icon on the top left and then choosing **Python Script**:



10. Write the code `a = [1, 2]` in the new script file, highlight it, and then click the **Run** icon on the top right:



You can see from the console that RStudio uses reticulate behind the scenes. Moreover, on the top-right panel, you can inspect the Python variables created after the execution of your code.

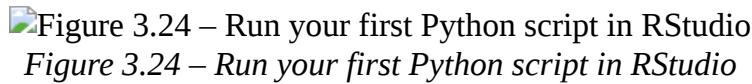
Great! You have successfully configured RStudio to run your Python code.

Configuring Python with Visual Studio Code

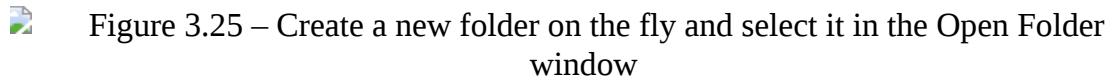
If you're an R language lover, chances are you'd prefer to run both R and Python code on RStudio. However, if you have the spirit of a true *Pythonista*, you'll definitely enjoy using one of the advanced editors that has been all the rage lately: **Visual Studio Code (VSCode)**. Let's install and configure it!

1. Download VSCode from this link (<https://code.visualstudio.com/>) by clicking on the **Download for Windows** button.
2. Run the executable and click **OK** when it asks to continue with the User Installer.
3. Accept the agreement and click **Next** on the next window.
4. Then keep the default destination folder and click **Next**.
5. Also keep the default Start menu folder on the following window and click **Next**.
6. On the **Additional Tasks** window, choose the tasks you prefer and click **Next**.

7. Click **Install** on the recap window.
8. After the installation is completed, keep the **Launch Visual Studio Code** checkbox flagged and click **Finish**.
9. After VSCode has started, click on the **Extensions** icon on the left, then start entering the `python` string and click on the **Python** extension:



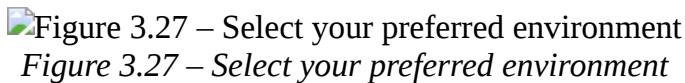
10. Click on **Install** on the extension's welcome page on the main panel.
11. Now go to the **File** menu at the top and click **Open Folder....**
12. Create a new test folder named **Hello** from the **Open Folder** windows and then select that folder:



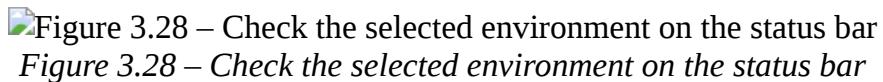
13. Now you have to select the Python interpreter accessing the **Command Palette** using `Ctrl + Shift + P`. Then start entering the `interpreter` string and click **Python: Select Interpreter**:



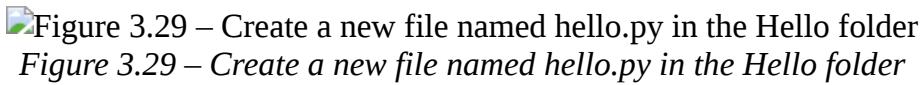
14. You'll be prompted to select one of the actual environments you have on your machine. Choose the **pbi_powerquery_env** environment:



15. You can see the selected environment on the status bar on the left:



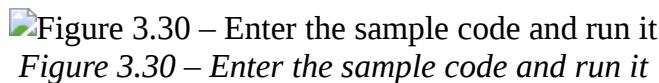
16. Go back to your **HELLO** folder in the Explorer panel on the left. Click on the *new file* icon next to the **HELLO** label and name the new file `hello.py`:



17. Enter the following code in the new file:

```
msg = "Hello World"  
print(msg)
```

Then click the *run* icon (the green triangle) on the top right of the main panel:



18. You can see the result in the following Terminal output:



Very good! Your VSCode is now configured correctly to run Python scripts.

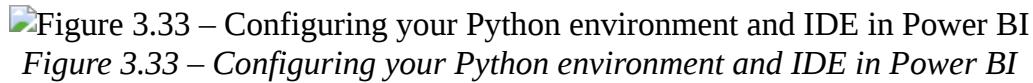
Configuring Power BI Desktop to work with Python

Since you have everything you need installed, you can now configure Power BI Desktop to interact with Python engines and IDEs. This is really a very simple task:

1. In Power BI Desktop, go to the **File** menu, click on the **Options and settings** tab, and then click on **Options**:



2. In the **Options** window, click on the **Python scripting** link on the left. The contents of the panel on the right will update, giving you the option to select the Python environment to reference and the Python IDE to use for Python visuals. By default, the detected interpreter is the default one installed by Miniconda. In order to select a specific environment, you need to choose **Other** and then click **Browse** and supply a reference to your environment folder:



Usually, you can find the default environments folder in `C:\Users\<your-username>\miniconda3\envs\`. Then select your `pbi_powerquery_env` subfolder.

3. By default, Power BI recognizes VSCode as a Python IDE. Keep it as is and click **OK**.

You will see how to interact with the IDE from Power BI Desktop when we introduce the R and Python script visuals in *Chapter 12, Exploratory Data Analysis*.

Configuring the Power BI service to work with R

As you have already learned from *Chapter 2, Configuring R with Power BI*, in order to allow the Power BI service to use R in the data transformation steps with Power Query, you must install the **on-premises data gateway in personal mode** on an external machine, on which an R engine is installed. The same thing applies to Python with Power Query in the Power BI service. So, if you have not installed the on-premises data gateway yet, do it by following the steps in *Chapter 2*.

Important note

Python engines and R engines can be *installed on the same external machine* and *referenced by a single data gateway*. You must make sure, however, that the machine's resources are sufficient to handle the load of requests coming from the Power BI service.

Sharing reports that use Python scripts in the Power BI service

What was said about how to share reports that use R scripts for data transformations in the Power BI service in *Chapter 2, Configuring R with Power BI* also applies to reports that use Python scripts. Consider the following tip in summary.

Tip

You can use an unofficial architecture that makes use of a personal data gateway associated not with a physical person, but with a *fictitious "service" user*. The credentials of this user are shared between all those analysts who use Python (along with R) code to transform data into their reports. Also, since the machine with the Python engines referenced by the gateway is shared, it must remain on during periods of scheduled activity. The same machine often hosts the R engines too. An *Azure Windows Virtual Machine*, on which the R and Python engines and the data gateway run, is often used in this architecture.

As a reminder, *Figure 3.34* summarizes the aforementioned architecture:



Figure 3.34 – Enterprise architecture for the use of Python and R in data transformations
Figure 3.34 – Enterprise architecture for the use of Python and R in data transformations

Thanks to this architecture it is possible to allow a group of analysts to be able to use Python scripts in their reports, despite the limitations imposed by the on-premises data gateway in personal mode.

That said, in addition to the limitations seen for Python scripts in Power Query, there are some important ones to be aware of for Python visuals as well.

Limitations of Python visuals

Python visuals have some important limitations regarding the data they can handle, both input and output:

- A Python visual can handle a *dataframe with only 150,000 rows*. If there are more than 150,000 rows, only the first 150,000 rows are used.
- Python visuals have an *output size limit of 2MB*.

You must also be careful not to exceed the *5-minute runtime calculation* for a Python visual in order to avoid a time-out error. Moreover, in order not to run into performance problems, *the resolution of the Python visual plots is fixed at 72 DPI*.

As you can imagine, some limitations of Python visuals are different depending on whether you run the visual on Power BI Desktop or the Power BI service.

When creating reports in *Power BI Desktop*, you can do the following:

- Install any kind of package (PyPI and custom) in your engine for Python visuals.
- Access the internet from a Python visual.

When creating reports in *the Power BI service*, you can do the following:

- You can only use the PyPI packages listed at this link: <https://bit.ly/powerbi-python-limits>.
- You cannot access the internet from a Python visual.

Important note

In contrast to the case of R visuals, you **do not** have the option of developing a custom Python visual.

Once one of your reports is published on the Power BI service, you can decide to share it on your blog, on one of your websites, or on social media via the **Publish to web** option.

Unfortunately, Python visuals (as well as R visuals) *are not usable on reports to be published publicly on the web by design by Microsoft*.

Tip

If you absolutely need to publish a particular visualization to the web, remember that *custom R visuals* overcome the limitation on publishing to the web. So, you need to switch from Python to R.

Summary

In this chapter, you learned about the most popular free Python distributions in the community and the best practices for their use.

Using the unique features of Power BI Desktop and the Power BI service, you have learned how to properly create specific Python environments.

You also learned that the most popular IDE in the R community (RStudio) can also run Python code. In addition, you have installed and configured VSCode, which is to date one of the most widely used advanced editors for Python.

You were also introduced to all of the best practices for properly configuring both Power BI Desktop and the Power BI service with Python, whether in a development or enterprise environment.

Finally, you've learned some of the limitations on using Python with Power BI, knowledge of which is critical to avoid making mistakes in developing and deploying reports.

In the next chapter, we'll finally start working with R and Python scripts in Power BI, doing data ingestion and data transformation.

4 Importing Unhandled Data Objects

In this chapter, you'll look at using R and Python in what is typically the first phase of report creation: **data ingestion**. Power BI is a very powerful tool from this point of view, because it has many connectors to various data sources out of the box. In addition to being able to import data directly by connecting to data sources, you can easily solve more complex data loading scenarios with Power BI. For example, you can merge multiple CSV files or multiple Excel Workbook's sheets dynamically directly from Power BI, even using the **M language** to apply special logic to the merge step. You can also scrape any web page by just clicking on web page contents without using any code. All this is possible thanks to Power BI's standard features, without having to use R or Python.

There are, however, cases in which the data to be imported and used in Power BI comes from **processing done on external systems**, which persist data in formats that are not directly managed by Power BI. Imagine being a Power BI report developer and having to interface with a team of data scientists. Some complex processing done by them on fairly large datasets might require non-trivial run times. That's why data scientists often **serialize the result** of such processing in files of an acceptable size, so they can deserialize them very quickly if needed. Now suppose the data scientist team provides you with one of these files serialized in R or Python and asks you to use it for some calculations needed to create a visual in your report. How would you do it?

In this chapter, you will see how to work with serialized files from R (.rds) and Python (.pk1) in Power BI. The following topics will be discussed in this chapter:

- Importing RDS files in R
- Importing PKL files in Python

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

Importing RDS files in R

In this section, you will develop mainly R code, and in the various examples, we will give you an overview of what we are going to do. If you have little experience with R, you should familiarize yourself with the data structures that R provides by starting with this quickstart: <http://bit.ly/r-data-struct-quickstart>. Take a look at the *References* section for more in-depth information.

A brief introduction to Tidyverse

A data scientist using R as an analytical language for data analysis and data science must know the set of packages that goes by the name of **Tidyverse** (<https://www.tidyverse.org>). It provides everything needed for data wrangling and data visualization, giving the analyst a consistent approach to the entire ecosystem of packages it provides. In this way, it tries to heal the initial situation of "chaos" of R functionalities provided by packages developed by developers who had not agreed on a common framework.

Note

If you are new to R, you might want to start with this quickstart by Software Carpentry to get familiar with the main concepts: <http://bit.ly/tidy-quickstart>. The *References* section also contains links to in-depth information about Tidyverse.

The fundamental data type to know about in Tidyverse to be able to work with tabular data is the **tibble**. Tibbles (the New Zealand way of pronouncing "tables") are a modern version of R **dataframes**. Starting from a tibble,

you can perform all the data transformations you want with simple functions provided by the Tidyverse packages.

Today, you will often find the use of the **%>% pipe** (the R language allows symbols wrapped in % to be defined as functions and the > implies a chaining) in data analyses performed in the Tidyverse world. Borrowed from the **magrittr** package included in Tidyverse, the pipe has the function of forwarding the object on its left inside the function on its right as the first parameter. In short, if you need to select the `my_col` column from a `my_tbl` tibble, instead of using `select(my_tbl, my_col)`, you can use the piped form `my_tbl %>% select(my_col)`, making the code much more readable.

Important Note

Currently, R Core is planning to introduce a new graphical form of the pipe, which is |>. So, be ready to use it when they make it available.

For the purpose of summarily understanding the code used in this section, we will describe it piece by piece, explaining the functionality of each R object used.

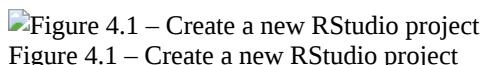
Creating a serialized R object

Now imagine for a moment that you are part of the data scientist team that has to do the complex processing of a dataset and then serialize the result obtained in a file to be reused as needed. The first thing to do is to configure the environment to install the latest version of Tidyverse.

Configuring the environment and installing Tidyverse

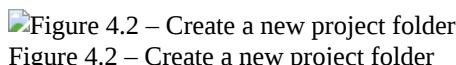
Open RStudio and proceed as shown here:

1. Make sure the most recent R engine (4.0.2 in our case) is selected (**Tools** and **Global Options...**).
2. Create a new project by clicking on the **Project** icon in the upper-right corner and then **New Project...**:



An RStudio project makes it straightforward to divide your work into multiple contexts, each with its own working directory, workspace, history, and source documents.

3. Click on **New Directory** and then on **New Project**.
4. Now enter a name for the project folder, choose in which folder do you want to place it, and click **Create Project**:



You can also find this project ready for you in the GitHub repository here: `Chapter04\importing-rds-files\importing-rds-files.Rproj`. RStudio will restart the R session and the project folder you have just created will become the working directory of your project.

5. If you recall, back in *Chapter 3, Configuring Python with Power BI*, you already disabled the MRO restriction on using a snapshot made at an earlier date from which to download packages. This was meant to install the latest versions of the packages. The problem is that the effect of that operation was temporary. In order to see that, run the `getOption("repos")` command in the console right now. You'll see that the default repository is still the snapshot set by MRO.
6. In order to permanently override the repository at the project level, you must write the same code you used previously inside an `.Rprofile` file located in the project folder. To do this, go to the console and type

`file.edit(".Rprofile")`. This will create the `.Rprofile` file in the project folder if it does not exist, and will open it in an editor window in RStudio. At that point, enter the following code:

```
local({r <- getOption("repos")
       r["CRAN"] <- "https://cloud.r-project.org/"
       options(repos = r)
     })
message("Default repo replaced with 'https://cloud.r-project.org/'")
```

Now save the `.Rprofile` file by pressing *Ctrl + S* (or **File | Save**) and then close the project by clicking on the **Project** icon (a cube containing “R”) in the upper-right corner, and then click **Close Project**:

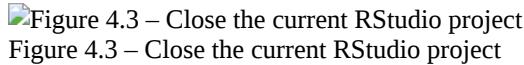


Figure 4.3 – Close the current RStudio project

Figure 4.3 – Close the current RStudio project

7. Reopen the project you just closed (you can find it in the list in the usual project menu at the top right, or you can find the `.Rproj` file in the project folder). You will notice that the message *“Default repo replaced with ‘https://cloud.r-project.org/’”* will appear in the console.
8. Type the `getOption("repos")` command again in the console and press *Enter*. Now you’ll see the new CRAN repository as the default one:



Figure 4.4 – The new CRAN repository set as default

Figure 4.4 – The new CRAN repository set as default

9. Now let’s install all Tidyverse packages by simply running the following command on the console:
`install.packages("tidyverse")` (it’s equivalent to installing it through the GUI by clicking on the **Packages** tab at the bottom right and then on **Install**).

Awesome! Now you are sure that you have installed the latest version of Tidyverse.

Creating the RDS files

We will now walk you through creating the serialization of an R object in an RDS file. Assume that the computational time required to create this object is very large. We will also have you create an object that is not a simple tibble, which could have been easily exported first to CSV and then imported into Power BI. Let’s start:

1. Open a new R script in your project by pressing *Ctrl + Shift + N* (or by clicking the **+ New File** icon and then **R Script**).
2. Now you will load in memory both the Tidyverse packages (using the `library` command) and the *growth population* tibble contained in the `tidyverse` package (using the `data` command), consisting of a subset of data taken from the *World Health Organization Global Tuberculosis Report*:

```
library(tidyverse)
data("population")
# Let's have a look at the tibble
population
```

The latest command (which matches the name of the tibble) allows you to observe the contents of the first few lines of the tibble and its columns’ data types. Highlight all the commands and press *Ctrl + Enter* (or click on the **Run** icon on the top right of the panel) to run them. You will see the following result:

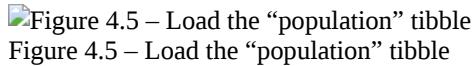


Figure 4.5 – Load the “population” tibble

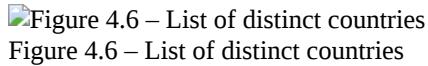
Figure 4.5 – Load the “population” tibble

Everything that follows `#` is a comment.

3. Now let's check how many distinct countries there are in the tibble. You'll use the `distinct` function and then the `pull` one in order to extract the single column of distinct countries from the tibble in vector format:

```
population %>%  
  distinct(country) %>%  
  pull()
```

You'll see a list of all the distinct countries, like this one:

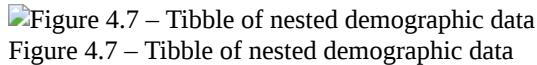


Try highlighting the code only up to and including `distinct(country)` and running the highlighted code. You will always see distinct countries, but still as part of a tibble.

4. Now we want to group the year and population information into separate tibbles for each country. In short, we want to have a tibble that contains the countries and for each of them another tibble with the demographic information by year. You can do that using the `nest` function:

```
nested_population_tbl <- population %>%  
  tidyr::nest( demographic_data = -country )  
nested_population_tbl
```

You have just assigned the `nested_population_tbl` variable the new tibble containing the nested demographic data. Observe that we made the `nest` function call explicit by calling it from its `tidyR` source package using `::`. Also, observe how easy it is to "nest" everything except the country column into a list of tibbles contained in the new `demographic_data` column. Highlighting and running the previous chunk of code, you'll see the following result:



Note that the new `demographic_data` column is a list of tibbles.

5. Now you can finally serialize the `nested_population_tbl` object into an RDS file using the `saveRDS` function:

```
saveRDS(nested_population_tbl, "nested_population_tbl.rds")
```

You can find the R code shown here in the GitHub repository associated with the book in the `Chapter04\importing-rds-files\01-create-object-to-serialize.R` file. To properly execute the code contained in the file, you should first open the RStudio project, double-clicking on the `Chapter04\importing-rds-files\importing-rds-files.Rproj` file. Then you can open the previously mentioned R script using the **Files** tab in the bottom-right panel of RStudio.

Awesome! You can verify that a file has been serialized correctly by taking a look at the same panel:



In the same way, you'll create an RDS object that contains time series views for four selected countries. The time series data is the same as the population growth data you saw earlier. Let's see how you can generate this file:

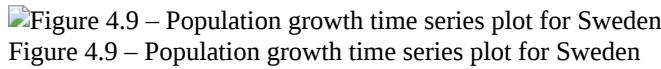
1. Install the fantastic **timetk** package by Matt Dancho by entering the `install.packages("timetk")` command into the RStudio console. It makes it easy to visualize, wrangle, and feature engineer time series data for forecasting and machine learning predictions. For more details, take a look here: <https://business-science.github.io/timetk/>.
2. Open the `Chapter04\importing-rds-files\04-create-plots-object-to-serialize.R` file in RStudio. The first part of the file contains code already seen in the previous section and is used to generate the nested

tibble of the population.

3. Immediately after creating the nested tibble, you'll see how to plot the time series data related to country Sweden. Every single R function used is explained in the code:

```
selected_country <- "Sweden"
nested_population_tbl %>%
  # Get the row related to the selected country
  filter( country == selected_country ) %>%
  # Get the content of 'demographic_data' for
  # that row. Note that it is a list
  pull( demographic_data ) %>%
  # Extract the 'demographic_data' tibble from
  # the list (it has only 1 element)
  pluck(1) %>%
  # Now plot the time series declaring the date variable
  # and the value one.
  timetk::plot_time_series(
    .date_var = year,
    .value = population,
    .title = paste0("Global population of ", selected_country),
    .smooth = FALSE,      # --> remove the smooth line
    .interactive = FALSE # --> generate a static plot
  )
```

Here is the result:



4. You will now create a time series graph for each country in the nested tibble following the previous example. The great thing is that thanks to the power of **functional programming** provided by the **map functions** of the **purrr** package, you can do this in one go using only one function. As always, you'll find detailed explanations in the code:

```
nested_population_plots_tbl <- nested_population_tbl %>%
  # Select a subset of countries
  filter( country %in% c("Italy", "Sweden", "France", "Germany") ) %>%
  # Add a new column called 'plot' applying the plot_time_series
  # function to the values of the demographic_data tibble (.x)
  # for each country (.y) in the 'country' field.
  # Do this thanks to the map2 function.
  mutate( plot = map2( demographic_data, country, ~ timetk::plot_time_series(.data = .x,
    .date_var = year,
    .value = population,
    .title = paste0("Global population of ", .y),
    .smooth = FALSE,
    .interactive = FALSE) )
) %>%
# Return just the 'country' and 'plot' columns.
select( country, plot )
```

5. After that, only the plots list extracted by the nested_population_plots_tbl tibble is serialized in an RDS file:

```
# The index of list items corresponds to the country_id values          # into the s
plots_lst <- nested_population_plots_tbl$plot
# Serialize the list of plots
saveRDS(plots_lst, "plots_lst.rds")
```

Well done! You've serialized your R objects into RDS files. If you want to try deserializing them, you can follow the code in the `02-deserialize-object-from-rds.R` and `05-deserialize-plots-object-from-rds.R` files you can found in the GitHub repository.

You are now ready to use the nested tibble serialized in a file directly in Power BI.

Using an RDS file in Power BI

It is clear that an RDS file must necessarily be used via R scripts in Power BI. As you may have learned by now, there are two Power BI objects through which you can use R scripts: **Power Query Editor** and **R visuals**. Let's start with the simplest case, which is to import the RDS file into Power Query Editor.

Importing an RDS file into Power Query Editor

You'll import a serialized R object into Power Query Editor when you know *you can extract tabular information from the object* and want to persist it in the Power BI data model. Let's see how it's done:

1. Go to RStudio and create a new R script into the project by pressing *Ctrl + Shift + N*. You could copy and paste the content of the `02-deserialize-object-from-rds.R` file (or open it directly if you used the GitHub `.Rproj` file to open the project).
2. Load the RDS file via the `readRDS` function and assign it to the new `deserialized_tbl` variable like so:

```
library(tidyverse)
project_folder <- "C:/<your>/<absolute>/<project_folder>/<path>"
deserialized_tbl <- readRDS( file.path(project_folder, "nested_population_tbl.RDS") )
```

Note that we are using an absolute path to read the RDS file, even though we are in an RStudio project and could have referenced the file without a path. This is because *Power BI does not have the concept of "projects" that RStudio does and therefore needs an absolute path* to locate the file correctly. Also, note that in R you can use either the double-backslash (\\\) or the simple slash (/) as a separator in path strings.

3. Now try extracting the demographics of Sweden from the nested tibble as follows:

```
sweden_population_tbl <- deserialized_tbl %>%
  filter( country == "Sweden" ) %>%
  pull( demographic_data ) %>%
  pluck(1)
sweden_population_tbl
```

In this piece of code, we are assigning to the `sweden_population_tbl` variable the content of the `deserialized_tbl` variable, to which we apply the following transformations:
a) We filter it for the country Sweden through the `filter` function (thus obtaining the row associated with the country Sweden).
b) From this row, we detach the content of the `demographic_data` field from the original tibble using the `pull` function (you'll get a list).
c) Since the content of the `demographic_data` column is a list containing only one tibble, the content must be unlisted using the `pluck` function. The result is the Sweden demographic data organized in one tibble, as shown in *Figure 4.11*:


Figure 4.10 – The content of the Sweden demographic data organized in a tibble

Figure 4.10 – The content of the Sweden demographic data organized in a tibble

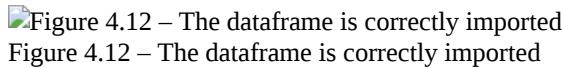
4. Now open Power BI Desktop and make sure it is referencing the earliest MRO. Click then on **Get data** and then **More....** Start typing `script` into the search textbox and double-click on R script. The R script editor will pop up.
5. Copy and paste the content of the `03-deserialize-object-from-rds-in-power-bi.R` file into the R script editor, changing the absolute path to the RDS file accordingly, and then click on **OK**.
6. The **Navigator** window will open, giving you the option to select which dataframe to import:


Figure 4.11 – Import the serialized dataframe into Power BI

Figure 4.11 – Import the serialized dataframe into Power BI

You will see as many dataframes (remember that tibbles are specializations of dataframes) as you have defined within your script. Select the `sweden_population_tbl` dataframe and click **Load**.

7. When loading is finished, click on the table icon on the left menu of Power BI Desktop to verify that the data has been imported correctly in tabular form:



Great! You have correctly imported your RDS file into Power BI to use its contents with Power Query in the most appropriate way.

Important Note

As you may have noticed, *the only R data structure that Power BI can work with are dataframes (or datatable) with columns that have standard data types*. It is not possible to import any other type of R objects. If you had imported the `deserialized_tbl` dataframe directly, the values in the `demographic_data` column would have generated an error and would have been unavailable.

Sometimes it may happen that you don't have the ability to deserialize an RDS file in Power Query Editor and extract its information in a tabular format in order to persist it in the Power BI data model. You may need to deserialize the content of an RDS file on the fly in an R visual in order to use its information in a visualization. You'll see how to solve this scenario in the next section.

Importing an RDS file in an R visual

Now suppose you have received the RDS file containing the time series charts for each country from the data scientists team. Your goal is to allow the report user to view the charts by selecting a country.

The problem you are facing is as follows: you know that in order to import any information from Power Query Editor via the R script, it must be in tabular format and must use standard data types. The charts made available by the data scientists are grouped in a list in the `ggplot` format (`ggplot` offers a powerful graphics language for creating elegant and complex plots in R), which in itself is not a standard data type. How do you import them into Power BI? You'll need a little bit of *lateral thinking*.

Important Note

As you probably already know, it is possible to serialize any programming object in its **byte representation** (raw vector in R). The byte representation can in turn be transformed into its **string representation**. Once you have strings (a standard data type), you can organize them into a dataframe. After that's done, you can import that dataframe into Power BI.

The moment you need to "feed" an R visual with data, keep the following considerations in mind:

Important Note

When you select more than one column from more than one table in the Power BI data model (there must be a relationship between them) as values of an R visual, these values *will form a single dataframe (deduplicated)* to be referenced in the R script of the visual.

Also, keep the following in mind:

Tip

In some cases, you may want to *not delete duplicate rows*. In that case, you can add an index field to your dataset (row number) that causes all rows to be considered unique and prevents grouping.

It wouldn't make sense to import a dataframe containing a string representation of something in Power Query Editor if you couldn't transform it back to the original object. Fortunately, the previously mentioned direct transformation operations are all invertible, so you can use the inverse transformations within an R visual to

extract and display plots appropriately. Also added to this process is a limitation of the data handled in R visuals that appears to be undocumented.

Important Note

If a string is longer than 32,766 characters, once passed into the default dataframe to be referenced within an R visual, *it is truncated*. To avoid truncation, it is necessary to split the string into smaller chunks (we chose an arbitrary length of 10,000) and persist those chunks in a dataframe column before using the data into the R visual.

That said, in summary, what you will be doing in this section is as follows:

1. Import the RDS file containing the named list of plots in **Power Query Editor**. From it, extract a dataframe of country names and a dataframe containing plots information.
2. Use the countries dataframe in a slicer with a single choice.
3. Each time a country is selected from the slicer, the **R visual** will display the plot of the time series of the population growth of that country.

Let's summarize all the processes of wrangling the plots data in a figure that contains the functions you will find in the code:

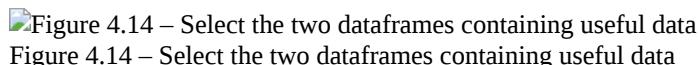


Important Note

It hasn't been possible to directly import the single dataframe containing both the country names and the corresponding plots, because the R engine returns a mysterious **ENVSXP Error**. A named list works like a charm.

In the following steps, we will not explain in detail all the functions used, simply because we will refer to the code shared in the GitHub associated with this book, in which every single detail is commented. So, let's start:

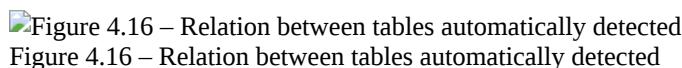
1. Open Power BI Desktop and go to **Get data**, then **More...**, then **R Script** to import the RDL files.
2. Open the `06-deserialize-plots-object-from-rds-in-power-bi.R` file from the GitHub repository, copy the content, changing the absolute path to the RDL file accordingly, paste it into the R Script editor, and click **OK**.
3. Power Query will detect three dataframes created into your script. Select only the **plots_df** dataframe (the one containing the string representation of bytes of plots), and the **selected_countries_df** one. Then click **Load**:



4. Click on the **Data** icon on the left ribbon and then click on the **Manage relationships** button:



5. The engine has automatically created the relationship between the two imported tables:



Click **Close**.

6. Go back to the report canvas clicking on the **Report** icon on the left ribbon:

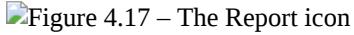


Figure 4.17 – The Report icon

7. Now click on the **Slicer** icon:

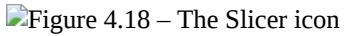


Figure 4.18 – The Slicer icon

8. Keeping the **Slicer** visual region selected into the canvas, click on the **selected_countries_df** table on the **Fields** panel, and select the **country** field:



Figure 4.19 – Select the country_name column for the Slicer visual

9. Then click the **Format** icon of the **Slicer** and enable the **Single select** option:



Figure 4.20 – Allow only a single selection

It is very important to set **Single select**, because *the logic inside the R visual will manage the deserialization of a single plot.*

10. Now the Slicer visual will show all the countries contained in the **selected_countries_tbl**:

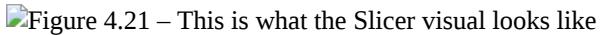


Figure 4.21 – This is what the Slicer visual looks like

11. Click on the report canvas in order to deselect the **Slicer** visual and click then on the **R script visual** icon:



Figure 4.22 – The R script visual icon

The usual **Enable script visuals** window pops up. Click on **Enable**.

12. Move and stretch the R visual borders in order to cover almost all the report canvas. Keeping it selected, click on the **plots_df** table into the **Fields** panel and select the **chunk_id**, **country_id**, and **plot_str** fields:

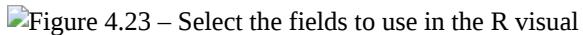


Figure 4.23 – Select the fields to use in the R visual

Feel free to turn the R visual title off in the **Format** tab.

1. Open the `07-deserialize-plots-df-into-r-visual.R` file from the GitHub repository, copy the content, and paste it into the R visual's script editor. Then click on the **Run** icon on the top right of the R script editor.
2. Now you can click on each country into the Slicer in order to see its population time series:



Figure 4.24 – Showing the population growth for Italy

Outstanding! You have just created a report that very few could have made.

Important Note

This technique can be very useful when you need to build complex visualizations in R that require the use of packages not provided by the R visual in Power BI Service. These visualizations can be made "offline," serialized to file, and then used on a shared report on the service.

You've just seen how to import an RDS file despite not being able to do so out of the box in Power BI, and how you can then use it directly within an R visual.

In the next section, you'll see how you can do the same thing for files serialized with Python.

Importing PKL files in Python

Let's give you an overview of what you're going to implement using the Python code on GitHub. If you are not familiar with Python, you should familiarize yourself with the basic structures through this tutorial:

<http://bit.ly/py-data-struct-quickstart>. For a more detailed study of how to implement algorithms and data structures in Python, we suggest this free e-book: <http://bit.ly/algo-py-ebook>.

A very short introduction to the PyData world

The **PyData** world is made up of users and developers who are passionate about data analytics and love to use open source data tools. The PyData community also loves to share best practices, new approaches, and emerging technologies for managing, processing, analyzing, and visualizing data. The most important and popular packages used by the Python data management community are as follows:

- **NumPy**: This is the main library for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with data.
- **Pandas**: The *Python Data Analysis Library* (pandas comes from *panel data*) is built upon NumPy and takes data in a tabular format (such as a CSV file, TSV file, or SQL database table) in order to create a Python object with rows and columns called a **DataFrame**. This object is very similar to a table in statistical software and people familiar with R conceptually equate the pandas DataFrame with R's dataframe data type.
- **Matplotlib**: This is a Python package used to produce plots. It began as a project in the early 2000s to use Python to visualize electronic signals in the brains of patients with epilepsy. The creator of Matplotlib was a neurobiologist and was looking for a way to replicate MATLAB's graphing capabilities with Python.
- **Scikit-learn**: Also known as **sklearn**, this derives its name from the fusion of the two words "SciPy" and "Toolkit." It is a free and robust machine learning library for Python designed to interact with the NumPy, SciPy, pandas, and Matplotlib, among others.

Going into detail about what is possible using these libraries is not the purpose of this book.

Note

If you want to start learning how to work with data in Python by taking advantage of these libraries, we recommend starting with this free book: <http://bit.ly/data-science-py-ebook>. After that, for further study, you can't miss the opportunity to study this fantastic book: *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-Learn, and TensorFlow 2, Third Edition*, by Sebastian Raschka, Packt.

In order to summarily understand the code used in this section, we will try to describe its functionality piece by piece, referring you to the comments in the code on GitHub for more details.

Creating a serialized Python object

As done in the previous section, now imagine that you are part of another team of data scientists that needs to do some complex, time-consuming data analysis with Python. Needing to reuse the results obtained in other processes, the team decides to use a **Pickle file** (PKL). It is obtained by serializing and then writing to a file any Python object, using the **pickle** library. As you have already seen in the previous section, serializing means converting an object in memory into a stream of bytes that can be either saved to disk or sent over a network. Obviously, this is an easily reversible operation. In fact, there is the possibility to deserialize a serialized object.

Important Note

Before you start, make sure that the path where you unzipped the GitHub repository ZIP file **doesn't contain any spaces in the names of the folders that make it up**, otherwise the Python script execution will give an error.

So, let's start to install what we need in our environment and initialize the IDE appropriately.

Configuring the environment and installing the PyData packages

Open your **Anaconda Prompt** (from the **Start** menu) and proceed as follows:

1. Make sure to use the `pbi_powerquery_env` environment, entering this code:

```
conda activate pbi_powerquery_env
```

Now you will install some missing packages that are necessary to be able to use the code in this section.

2. Enter the following command to install **matplotlib**: `pip install matplotlib`.

Great! Your Python environment is now ready to run your scripts. Now open Visual Studio Code and proceed as shown:

1. Go to **File** and then **Open Folder....** Make sure to choose the **importing-pkl-files** folder contained in the GitHub repository you previously unzipped, under the **Chapter04** folder. Click **Select Folder**.
2. Open the `01-create-object-to-serialize-in-pkl.py` file, clicking on it into the Explorer on the right, under the selected folder.
3. Remember you have to choose the environment in which to run your script. So, press *Ctrl + Shift + P* to open the Visual Studio Code palette and start entering the text “interpreter.” Then, select **Python: Select Interpreter**, and then choose the `pbi_powerquery_env` environment.

Excellent! Now you are ready to serialize your first Python object.

Creating the PKL files

Two of the most commonly used data structures in Python are **lists** and **dictionaries**. While by now you're familiar with lists, which you've seen before in R, if you've never developed in a programming language, perhaps dictionaries might sound new to you. Dictionaries are data structures that consist of a collection of **key-value pairs**. You can define them using curly braces (`{...}`). Specifically, in this section, you will create a dictionary with key-value pairs that consists of the country name and a dataframe containing data about the growth of the country's population.

The data you will use is the same data you used in the previous section. This time, instead of loading it from a package in memory, you'll do it directly from a CSV file. Let's go:

1. Since the `01-create-object-to-serialize-in-pkl.py` file is already open in Visual Studio Code, just run the code via the green arrow icon in the upper-right corner (**Run Python File in Terminal**). This way, the whole script will be executed.
2. You won't see anything particular in the console, just the command that runs `python.exe` with the current script path as a parameter. But if you look in the explorer on the left, you will see that the `nested_population_dict.pkl` file has been created correctly:



3. Just like in Rstudio, you can only run pieces of code by highlighting them and pressing *Ctrl + Enter*. You need to change a settings option in order to allow the use of the **interactive window** with Python. Go to **Settings**, pressing *Ctrl + ,* (comma), then start entering **Send Selection To Interactive Window** in the search bar and check the selected option:

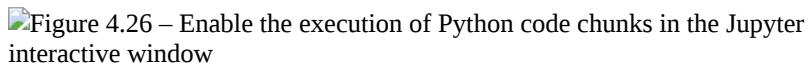


Figure 4.26 – Enable the execution of Python code chunks in the Jupyter interactive window

4. Now you have to install the *IPython kernel* (`ipykernel`) into your `pbi_powerquery_env` environment. Usually, this operation is done automatically by Visual Studio Code, but sometimes you can run into errors. So, it's better to do it manually. Open your Anaconda Prompt and enter the following command: `conda install --name pbi_powerquery_env ipykernel -y`.
5. Now select the code from the beginning (`import pandas as pd`) to the line where countries are defined (`countries = population_df.country.unique()`), then press *Shift + Enter*. Your chunk of code will be sent to the interactive window:

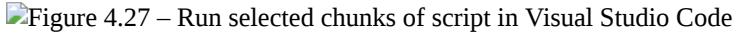


Figure 4.27 – Run selected chunks of script in Visual Studio Code

Clicking on the **Variables** icon into the interactive windows as shown in *Figure 4.28*, you can also inspect the content of each variable.

Hey, maybe you didn't notice, but with minimal effort, you have just created your first PLK file! You can train yourself to deserialize the newly created PKL file by running the code in the `02-deserialize-object-from-pkl.py` file.

Now we will now guide you in creating a second PKL file that contains a serialized dictionary with pairs composed of the country and the respective time series on population growth. This time, though, you will keep only four countries in the dictionary for simplicity. Let's proceed:

1. Open the `04-create-plots-object-to-serialize-in-pkl.py` file from the explorer on the left.
2. You can run the code a piece at a time to better understand how it works. About halfway through the script, you will find the following code:

```
# Let's try to plot the time series for Sweden
selected_country = "Sweden"
x = nested_population_dict[selected_country].year
y = nested_population_dict[selected_country].population
# Create a figure object
fig_handle = plt.figure()
# Plot a simple line for each (x,y) point
plt.plot(x, y)
# Add a title to the figure
plt.title("Global population of " + selected_country)
# Show the figure
fig_handle.show()
```

3. After running that piece of code, an interactive window will open in which the time series graph of Sweden is shown:



Figure 4.28 – Interactive window showing the time series plot of Sweden

Keep it open if you want to create new figures, otherwise you might get a weird error.

4. The last piece of code creates a new dictionary containing graphs for each country and serializes it to file. Once executed, you can see the `nested_population_plots_dict.pkl` file in the explorer on the left:

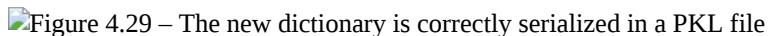


Figure 4.29 – The new dictionary is correctly serialized in a PKL file

Amazing! You serialized your second dictionary as well. You can practice deserializing it using the code in the `05-deserialize-plots-object-from-pkl.py` script.

Now you're ready to test your PKL files in Power BI, either in Power Query Editor or in Python visuals.

Using a PKL file in Power BI

It is clear that a PKL file must necessarily be used through Python scripts in Power BI. So, there are two Power BI objects through which you can use Python scripts: **Power Query Editor** and **Python visuals**. Let's start with the simplest case, which is to import the PKL file into Power Query Editor.

Importing a PKL file in Power Query Editor

You will import a serialized Python object into Power Query Editor once you know how to extract tabular information from the object and then persist it in the Power BI data model. Let's take a look at how to do this:

1. Open Power BI Desktop and make sure it is referencing the pbi_powerquery_env environment. Then click on **Get data** and then **More....** Start typing script into the search textbox and double-click on Python script. The Python script editor will pop up.
2. Open the 03-deserialize-object-from-pkl-in-power-bi.py file in Visual Studio Code and copy its content. Then paste it into the Python script editor in Power BI Desktop, changing the absolute path to the PKL file accordingly, and click **OK**.
3. The **Navigator** window will open, giving you the option to select which dataframe to import:



Figure 4.30 – Import the deserialized dataframe into Power BI

Select the **sweden_population_tbl** dataframe and click **Load**.

1. When loading is finished, click on the table icon on the left menu of Power BI Desktop to verify that the data has been imported correctly in tabular form:



Figure 4.31 – The dataframe is correctly imported

Nice job! You have correctly imported your PKL file into Power BI to use its contents with Power Query in the most appropriate way.

Important Note

As with R scripts, **the only Python data structure that Power BI can work with are pandas DataFrames with columns that have standard data types**. It is not possible to import any other type of Python object. That's exactly why you couldn't import the dictionary directly.

As should be clear to you by now from the previous section, it can happen that a PKL file doesn't contain information in tabular format that can be extracted in Power Query Editor. You may need to deserialize the contents of a PKL file directly within a Python visual in order to use that information to create a chart. You will see how to solve this scenario in the next section.

Importing a PKL file in a Python visual

Now let's assume that you received a PKL file containing the time series graphs for each country from the data scientists team. Your goal is to allow the report user to view the graphs by selecting a country.

The problem you face is the following: you know that in order to import any information into Power Query Editor via Python scripts, it must be in tabular format and must use standard data types. The charts provided by the data scientists are grouped in a dictionary in the **figure format of Matplotlib**, which itself is not a standard data type. So, how do you import the dictionary into Power BI? The same "trick" used with R in the previous section applies.

In summary, what you will do in this section is this:

1. Import the PKL file containing the dictionary of plots in Power Query Editor. Extract its keys (countries) and expose them in a dataframe. Use its byte stream representation to fill another dataframe.
2. Use the countries dataframe as a slicer with a single choice.
3. Each time a country is selected from the slicer, the Python visual will deserialize the byte stream into the input dataframe and it will display the plot of the time series of the population growth of that country.

Important Note

Also, in this case, when you select more than one column from more than one table in the Power BI data model (there must be a relationship between them) as values of a Python visual, these values *will form a single dataframe (deduplicated)* to be referenced in the Python script of the Visual.

In addition, the same suggestion as was made for the R dataframe input applies:

Tip

In some cases, you may want to *not delete duplicate rows*. In that case, you can add an index field to your Pandas dataset (row number) that causes all rows to be considered unique and prevents grouping.

Again, even the Python visuals add a size limitation to the data it imports that appears to be undocumented:

Important Note

If a string is longer than 32,766 characters, *it is truncated* once passed into the input dataframe of a Python visual. To avoid truncation, we need to split the string into chunks of 32,000 characters each (this is an arbitrary value chosen by us) and persist these chunks in a column of the dataframe before using the data in the Python visual.

Here is the process summarised in a figure that contains the functions you will find in the code:

Figure 4.32 – Deserialize the PKL file content into a Python visual
Figure 4.32 – Deserialize the PKL file content into a Python visual

In the following steps, we will not explain in detail all the Python functions used, simply because we will refer to the code shared in the GitHub repository associated with this book, where every single detail is commented. So, let's get started:

1. Open Power BI Desktop and go to **Get data**, then **More...**, and then **Python Script** to import the PKL files.
2. Open the `06-deserialize-plots-object-from-pkl-in-power-bi.py` file from the GitHub repository, copy the content, paste it into the Python script editor, changing the absolute path to the PKL file accordingly, and click **OK**.
3. Power Query will detect three dataframes created in your script. Select only the `plots_df` (the one containing the chunks of byte strings of each plot) and `selected_countries_df` (the one containing the country names) dataframes:

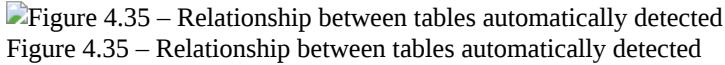
Figure 4.33 – Selecting the two dataframes containing useful data
Figure 4.33 – Selecting the two dataframes containing useful data

Then click **Load**.

4. Click on the **Data** icon in the left ribbon and then click on the **Manage relationships** button:

Figure 4.34 – The Manage relationships button
Figure 4.34 – The Manage relationships button

5. The engine has automatically created the relationship between the two imported tables:

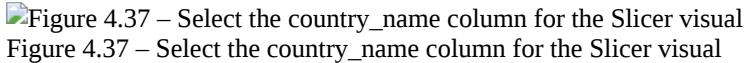


Click **Close** and go back to the report canvas using the **Report** icon in the left ribbon.

6. Now click on the **Slicer** visual icon:



7. Keeping the Slicer visual region selected into the canvas, click on the **selected_countries_df** table on the **Fields** panel and select the **country_name** field:



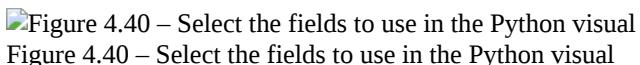
8. Then click the **Format** icon of the Slicer as you did in the previous section and enable the **Single select** option. The Slicer visual will show all the country names contained in the **selected_countries_df** table. It is very important to set **Single select**, because *the logic inside the Python visual will manage the deserialization of a single plot*.

9. Click on the report canvas in order to deselect the Slicer visual region and click on the **Python visual** icon:



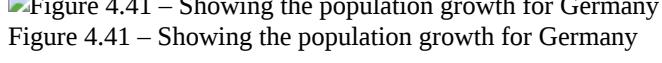
The usual **Enable script visuals** window pops up. Click on **Enable**.

10. Move and stretch the Python visual borders in order to cover almost all the report canvas. Keeping it selected, click on the **plots_df** table in the **Fields** panel and select all three **chunk_id**, **country_id**, and **plot_str** fields:



Feel free to turn the Python visual title off in the **Format** tab.

11. Open the `07-deserialize-plots-df-into-python-visual.py` file from the GitHub repository, copy the content, and paste it into the Python visual's script editor. Then, click on the **Run** icon on the top right of the Python script editor.
12. Now you can click on each country in the Slider in order to see the population time series:



Brilliant! You've just created a report using a methodology that few people in the world know about.

Important Note

This technique can be very useful when you need to build complex visualizations in Python that require the use of packages not provided by Python visuals in Power BI Service. These visualizations can be made offline, serialized to a file, and then used on a shared report on the service.

Summary

In this chapter, you got to learn about the Tidyverse approach to R development and how to serialize R objects to files. After that, you learned how to use these serialized files, both in Power Query Editor and in R visuals.

You then approached the same issues using Python. Specifically, you learned which packages are most used by the PyData community, learned how to serialize Python objects to files, and how to use them in Power BI, both in Power Query Editor and in Python visuals.

In the next chapter, you'll have a chance to learn how powerful regular expressions and fuzzy string matching are and what benefits they can bring to your Power BI reports.

References

For additional reading, check out the following books and articles:

- “*An Introduction to R*” by R Core (<https://cran.r-project.org/doc/manuals/r-release/R-intro.html>).
- “*R for Data Science*” by Hadley Wickham (<https://r4ds.had.co.nz/index.html>).
- “*Machine Learning with R: Expert techniques for predictive modeling, 3rd Edition*” by Brett Lantz, Packt Publishing (<https://www.packtpub.com/product/mastering-machine-learning-with-r-third-edition/9781789618006>).

5 Using Regular Expressions in Power BI

Often, many data cleansing tasks involve carrying out complex searches and substitutions between strings. The usual search and replace tools are sometimes not enough to get the desired results. For instance, let's suppose you need to match strings, not in an exact way (for instance, via equality conditions) but using similar criteria between them. Knowing how to use tools such as regular expressions (alias regex) or fuzzy string searches can make all the difference in projects that require high-quality data. Thanks to R and Python, you can add these tools to your arsenal.

In this chapter, we will cover the following topics:

- A brief introduction to regexes
- Validating data using regex in Power BI
- Loading complex log files using regex in Power BI
- Extracting values from text using regex in Power BI

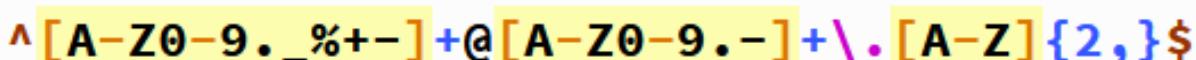
Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You need to properly configure the R and Python engines and IDEs, as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

A brief introduction to regexes

A **regular expression** (usually shortened to **regex**) is defined by a series of characters that *identify an abstract search pattern*. Essentially, it is a mathematical technique that was developed in 1951 by experts of formal language and theoretical computer science. It is used to **validate** input data or to *search for and extract* information from texts.

If you don't know the syntax of a regex, at first glance, it might look really tricky:



```
^ [A-Z0-9._%+-] + @ [A-Z0-9.-] + \. [A-Z] {2,} $
```

Figure 5.1 – An example of a regex pattern

Fortunately, there are online regex visualization tools that make it easier to understand patterns (you can find one of them at <https://regexpal.com>). For example, the regex highlighted in *Figure 5.1* can be visualized as follows:

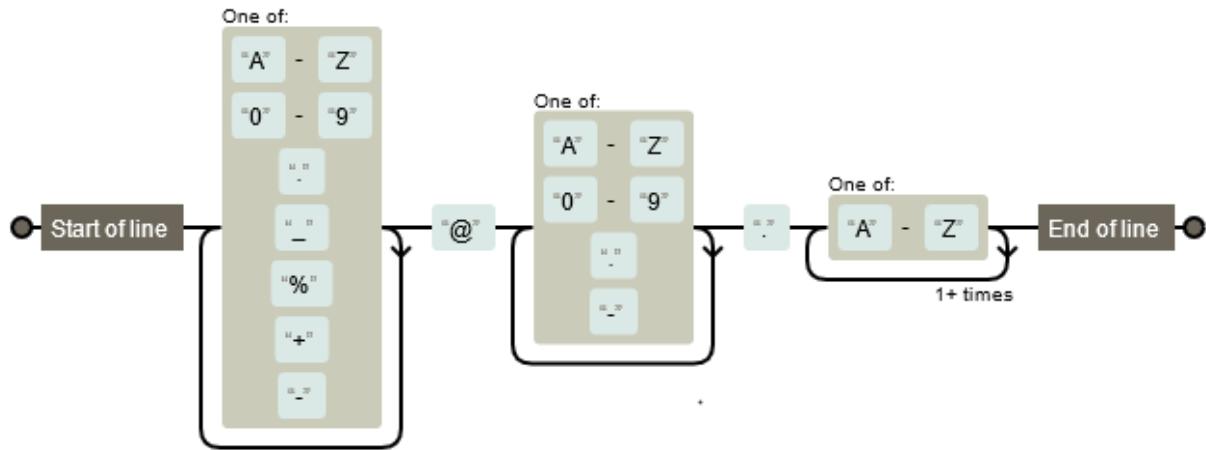


Figure 5.2 – A visualization of a regex

From *Figure 5.2*, it is enough to intuit that the regex in *Figure 5.1* will identify email addresses in a piece of text.

Learning how to use regexes like a pro is certainly not easy, and it is not the purpose of this section. Here, we will explain the basic rules that will allow you to create simple, yet effective, search patterns. For more details, please refer to the *References* section at the end of this chapter.

The basics of regexes

We will try to explain the basic principles of regexes through the use of examples, which is perhaps the most immediate way to start using them. Each subsequent subsection will explain a function of regexes. To test our regex, we will use the tool made available at <https://www.regexpal.com/>. Let's get started!

Literal characters

To include one or more literal characters in a regex, it is necessary to make use of the "search" feature. Let's try searching for the *owe* string inside the *May the power of extending Power BI with Python and R be with you!* text:

Figure 5.3 – Searching for "owe" using a regex

Note that the tool uses the **Global search flag** in the search by default. A specific flag is indicated with a letter (in our case, **g**) right after the regex delimiters, `.../`. The possible flags that can be used are as follows:

- **g (global)**: This will match all of the occurrences, keeping the index of the last match.
- **m (multiline)**: When enabled, the string anchors (you'll see them later) will match the start and end of a line instead of the whole string.
- **i (ignore case)**: This searches the pattern irrespective of the case (lower or upper) of the string.

Bear in mind that not all programming languages use flag syntax, as mentioned earlier. For example, Python's `re` package (the default one for regexes) provides parameters in the `search`, `match`, and `sub` functions:

```
re.search('test', 'TeSt', re.IGNORECASE)
re.match('test', 'TeSt', re.IGNORECASE)
re.sub('test', 'xxxx', 'TeSting', flags=re.IGNORECASE)
```

This is the same for R's `regex()` function of the `stringr` package:

```
str_detect('tEsT this', regex('test', ignore_case=TRUE))
```

You can also use **global modifiers** directly in line with your regex pattern. These are `(?i)` for case-insensitive and `(?m)` for multiline. For example, in R, you can also run the following script:

```
str_detect('tEsT this', regex("(?i)test"))
```

Note that Python doesn't allow inline global modifiers.

Special characters

Regex uses 12 special characters (also called **metacharacters**) where each has a special meaning. They are the pipe, `|`; the backslash, `\`; the dollar sign, `$`; the question mark, `?`; the caret, `^`; the asterisk, `*`; the plus sign, `+`; the dot, `.`; the parentheses, `(` and `)`; the opening square bracket, `[`; and the opening curly bracket, `{`.

If you need to search for one of the previously mentioned characters, you have to escape it using the backslash. So, if you want to match exactly `123$`, you need to use `123\$` as the regex pattern.

Next, you will learn about the meaning and use of metacharacters.

The ^ and \$ anchors

Anchor characters are special characters in that they are used to place the regex match at a certain position in the string. The caret, `^`, is used to indicate *the beginning of the string* (or line), and the dollar sign, `$`, is used to indicate *the end of the string* (or line). An example visualization is worth a thousand words:

The screenshot shows a regex search interface. In the 'Regular Expression' input field, the pattern `/may the power/i` is entered. A red arrow points from this field to the 'flags' dropdown menu. The 'flags' menu is open, showing the option `2 matches`. Another red arrow points from the 'flags' menu to the '2 matches' button. Below the input fields, there is a 'Test String' input field containing the text: `May the power of extending Power BI with Python and R be with you and may the power be endless!`. Both occurrences of the word 'may' are highlighted in blue, demonstrating the global search functionality.

Figure 5.4 – Case-insensitive and global search

In Figure 5.4, the "ignore case" flag is set by clicking on the **flags** icon and then checking "ignore case." In this way, both the occurrences are matched. Now, add a caret, `^`, before the `m` character:



Figure 5.5 – Case-insensitive and global search using the caret, ^

In this case, only the first occurrence (that is, the one at the beginning of the string) is matched.

If you also add a dollar sign at the end of the regex, nothing will be matched, as you are asking for a match of the `may the power` string that is at the beginning and that also ends the text.

OR operators

You might need to match one of the individual sets of characters or strings. For example, to match any of the s and t characters after the ye string, you should use the **character class** of [st] inside the `ye[st]` regex. This is so that it will match both the yes and yet strings. Character classes can also be used to match an occurrence in a range of characters using the hyphen, -. For example, `[0-9]` matches a single digit between 0 and 9, while `[A-Z]` matches a single uppercase letter from A to Z. Additionally, you can combine multiple ranges into one character class. For instance, `[A-Z0-9]` matches only a digit or an uppercase letter.

In order to match one of two strings, you can use the pipe, | , to separate them within opening and closing parentheses, such as `(string1|string2)`. Here is a complete example:

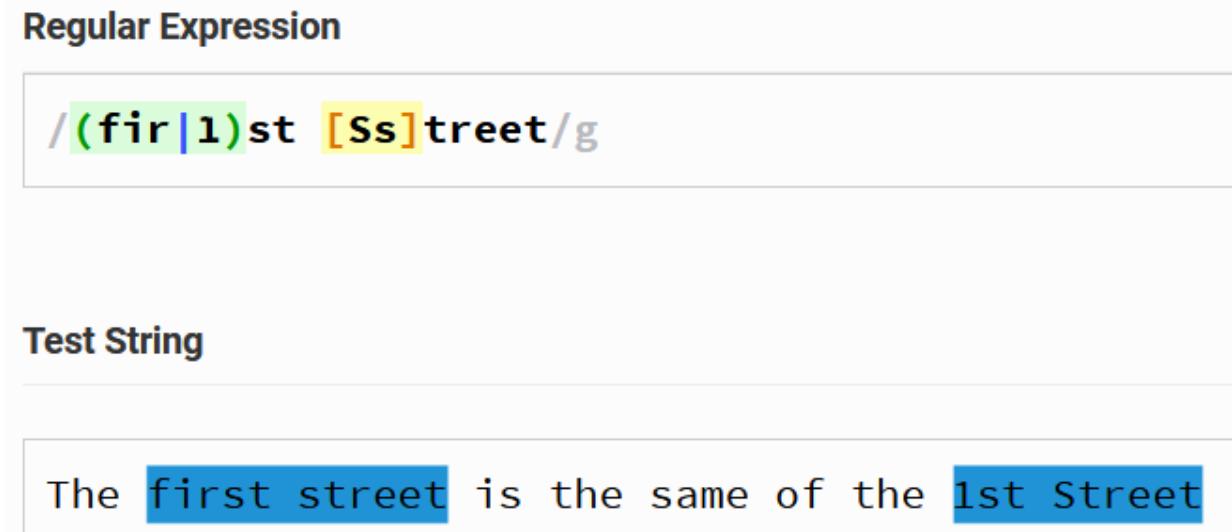


Figure 5.6 – A complete example of OR operators

The character class can also be used to match any character different from a specific character that is given. This is thanks to the **Negated Character Class**.

Negated character classes

The caret that appears just after the opening square bracket negates the content of the character class. For example, the `[^"]` regex matches every character that isn't a double quote.

Shorthand character classes

There are some character classes that are used very often. For this reason, we have defined some abbreviations to allow you to include them in regexes quickly. Here is a list of the most used ones:

- `\w` : This matches an alphanumeric character, including the underscore.
- `\W` : This is the opposite of `\w`, so it matches a single non-alphanumeric character, excluding the underscore. For example, it can match spacing and punctuation marks.
- `\d` : This matches a single digit.
- `\D` : This is the opposite of `\d`, so it matches a single non-digit character.
- `\s` : This matches "whitespace characters" such as space, tab, newline, and carriage return.

We'll use shorthand character classes relatively often throughout this chapter.

Quantifiers

Quantifiers indicate the number of times a *character* or *expression* must be matched. Here is a list of the most used ones:

- `+` : This matches what precedes it one or more times. For example, `test\d+` will match the `test` string followed by one or more digits. The `test(-\d\d)+` regex will match the `test` string followed by one or more times a dash that is then followed by two digits:

Regular Expression

```
/test(-\d\d)+/g
```

Test String

```
test-01 is different from test-23-456
```

Figure 5.7 – Repeating a group of characters using `+`

- `{n}` : This matches what precedes it *n* times. For example, `\d{4}` will match any integer number made up of 4 digits.
- `{n, m}` : This matches what precedes it between *n* and *m* times. For example, `prod-\d{2, 6}` will match the `prod-` string followed by an integer number made up of between 2 and 6 digits.
- `{n, }` : This matches *n* or more times what precedes it.

- ? : This matches one or zero times what precedes it. For example, Mar(ch)? will match both March and Mar . Alternatively, the colou?red regex will match both colored and coloured .
- * : This matches zero or more times what precedes it. For example, code\d* will match code , code1 , or code173846 .

The dot

The dot corresponds to a single character, regardless of what that character is, except for line break characters. It's a very powerful regex metacharacter and gives you a chance to be lazy. This is precisely why you have to be careful not to abuse it because, sometimes, you might include unintended results in the matches.

Greedy and lazy matches

The +, *, and repetition of {...} are **greedy quantifiers**. Greedy means that *they will consume the longest possible string*. Let's suppose that you only want to match the tags used in the Power BI rocks string. The first attempt a beginner would make is to use the <.+> regex, which, expressed in words, becomes "get the <, then get any non-newline character one or more times, and finally, in the end, get the >." The expected result is made by two matches, and . Let's take a look at the result:

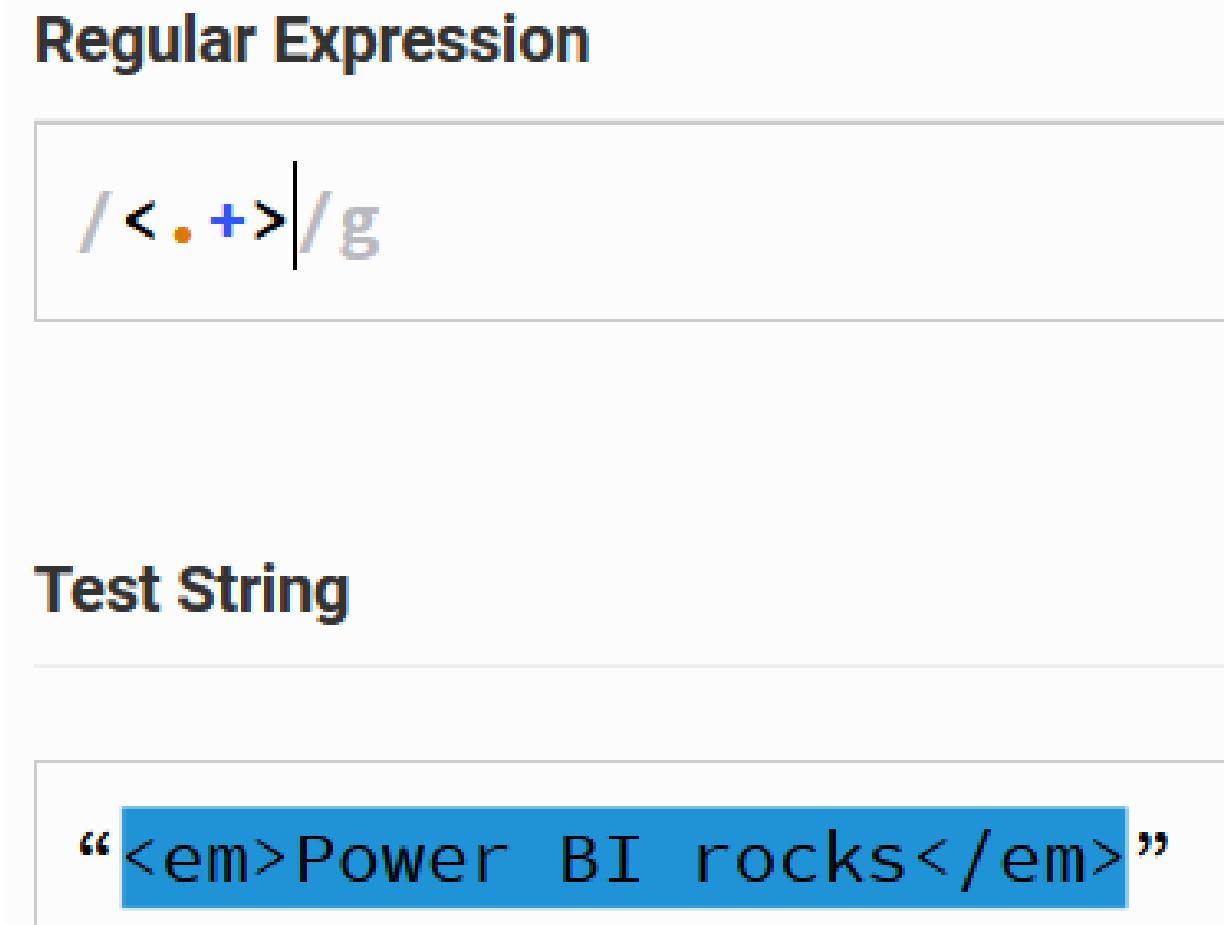


Figure 5.8 – The greediness of .+

It is evident that the combination of .+ captures everything contained between *the first occurrence of <* and the last occurrence of *>* , hence the definition of the **greediness** of the quantifiers.

So, is it possible to force a greedy quantifier to stop at the first detected occurrence of the next character, preventing it from "eating" anything until the last occurrence of the same? In other words, is it possible to turn a greedy quantifier into a **lazy** one? The answer is "yes," it is possible to do so by adding the `?` metacharacter just after the `+`. So, the `<.+>` regex becomes `<.+?>`. Here is the result:

Regular Expression

```
/ < . + ? > /g
```

Test String

```
“<em>Power BI rocks</em>”
```

Figure 5.9 – Making a greedy quantifier lazy thanks to the `?` metacharacter

Bear in mind, however, that *lazy quantifiers are underperforming*. Whenever possible, it is always preferable to *use negated character classes instead*. In our example, simply using the `<[^>]+>` regex (that is, a `<` character, any non-`>` character one or more times, and a `>` character) will achieve the same result without consuming computational resources:

Regular Expression

```
/<[^>]+>/g
```

Test String

```
"<em>Power BI rocks</em>"
```

Figure 5.9 – Using a negated character class instead of a lazy quantifier

So far, with what you've learned about regexes, you have the minimum foundation required to understand the more complex regexes that we'll be using in the next few examples.

Checking the validity of email addresses

If you were asked to validate an email address using the concepts you just learned, one of your first attempts might look like the following: `^.+@.+\.+$.+`. Translating this regex into spoken language gives us the following:

1. `^`: This matches the beginning of the string or a line if the multiline flag is enabled.
2. `.+`: This matches any character one or more times, except the line break.
3. `@`: This matches an "@" character.
4. `.+`: This matches any character one or more times, except the line break.
5. `\.`: This matches a "." character.
6. `.+`: This matches any character one or more times, except the line break.

Of course, this regex will validate a correct email address. But are you sure it can also detect the obvious syntactic errors of bad emails? Let's perform a test in <https://www.regexpal.com/> with the wrong email, `example@example.c` (the top-level domain, that is, the portion of the domain after the dot, must contain a minimum of two characters):

Regular Expression

```
/^ .+@ .+ \. .+ $/g
```

Test String

```
example@example.c
```

Figure 5.10 – Using a simple regex to validate a wrong email address

Well, that's not much of an outcome: an obviously wrong email would pass as correct. For this reason, it is often necessary to use more complex regexes that can respect well-defined syntactic rules.

In this specific case, we'll use a specific regex for email validation that we often adopt in production. It also takes into account whether the domain IP is used. For the purpose of displaying it in full, the regex is as follows:

```
^(([^\<()"\.,;:\s@"]+(\.\[^\<()\"\.,;:\s@"]+)*|(\\".+\\")|((\?[\0-9]{1,3}\.[\0-9]{1,3}\.
```

At first glance, this regex is sure to cause confusion. However, if we attempt to break it down into its essential parts, it becomes much more readable.

However, if you think that this regex is really complex, take a look at the one that takes into account all, and I mean all, the syntactic rules provided by the *Standard for the Format of Arpa Internet Text Messages*, which you can find at <http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html>! Pretty impressive, huh?

The format of an email address is defined as *local-part@domain*. You can find the complete specifications on Wikipedia at https://en.wikipedia.org/wiki/Email_address. We are going to match the minimum number of rules (not all of them!) that will allow us to validate a significantly different number of email addresses. So, considering we're going to match the domain name or the domain IP, the general structure of the regex is as follows:

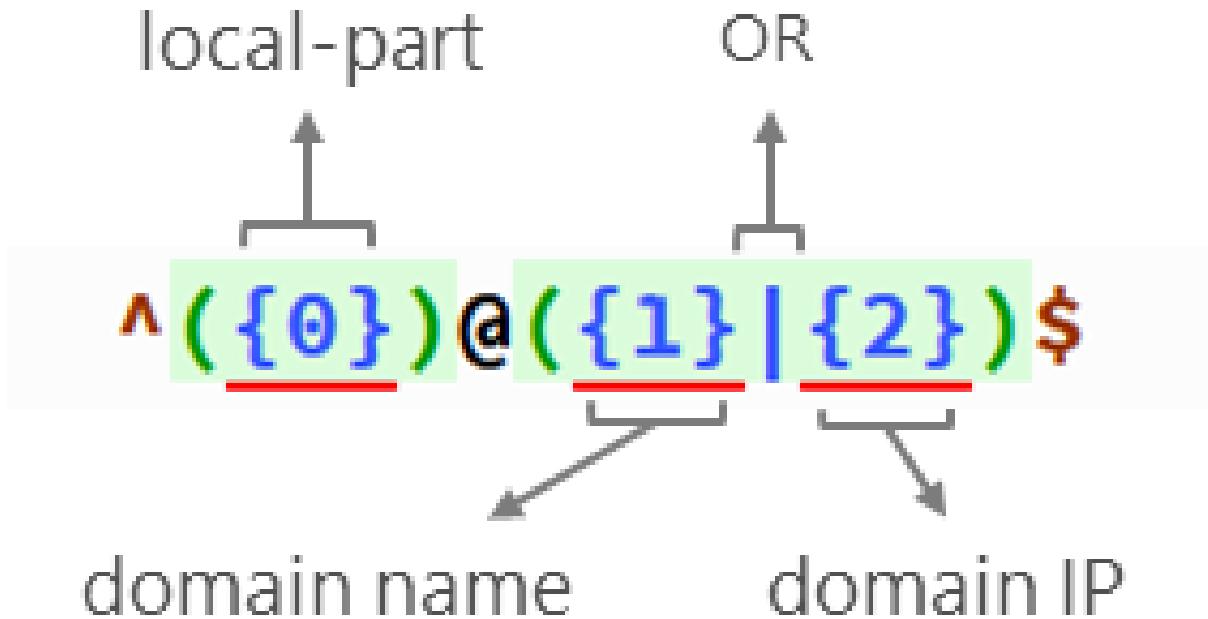


Figure 5.11 – The structure of the complex regex for email validation

In Figure 5.11, the `{0}`, `{1}`, and `{2}` strings are just placeholders, not characters to match. That said, let's start by defining each token:

1. The `{0}` token matches the **local-part** regex of the email. In this case, it's much easier to explain what the **local-part** regex does with a diagram rather than with words. In Figure 5.12, the labels explain every single detail of the subparts:

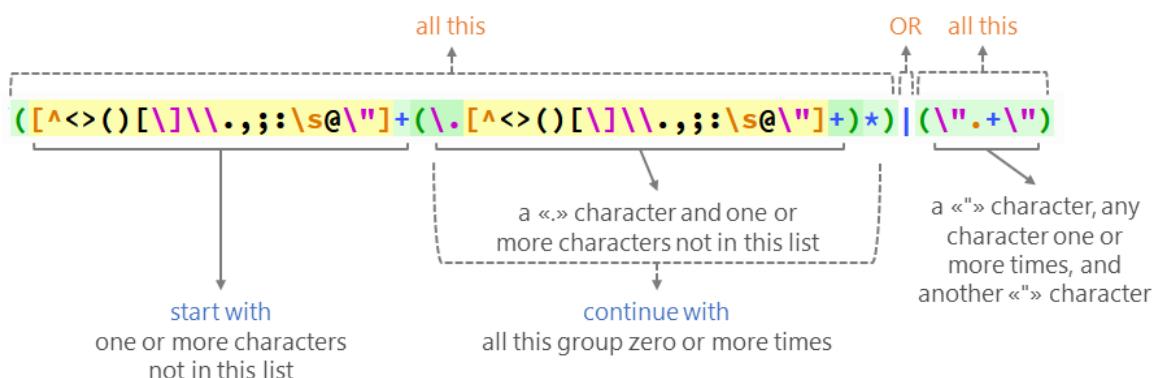


Figure 5.12 – The local-part regex explained in detail

Bear in mind that *parentheses group regex patterns together*. Parentheses allow you to apply regex operators to the entire grouped expression and to get a part of the match as a separate item in the result array. The text that corresponds to the regex expression is captured within them as a **numbered group**, and it can be reused

with a numbered backreference. You'll learn how to use this feature later. Remember that, in order to have the match of a metacharacter be a real character, the escape backslash must be placed before it. So, for example, `\]` will be the `]` character.

2. The `{1}` token matches the **domain name** of the email. Again, we will use a diagram to explain what the domain name regex does:

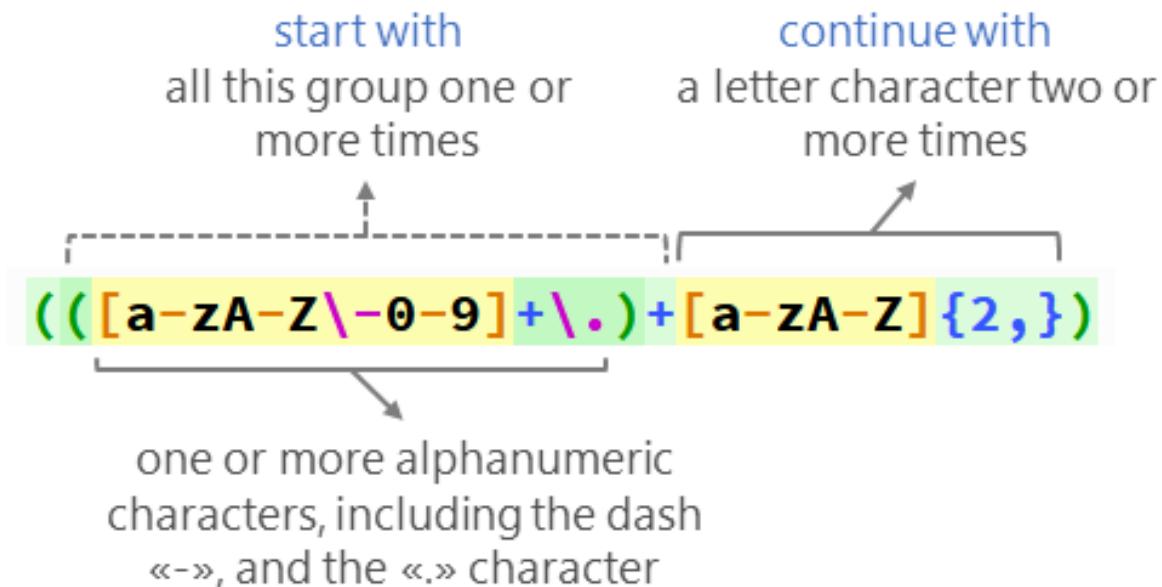


Figure 5.13 – The domain name regex explained in detail

3. The `{2}` token matches the **domain IP** of the email. It is the easier sub-regex, and you can find out what it matches by looking at *Figure 5.14*:

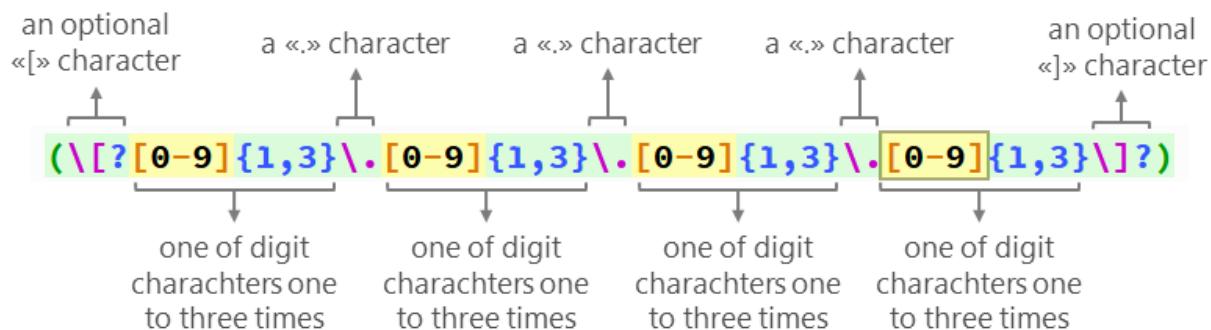


Figure 5.14 – The domain IP regex explained in detail

If you want to view the whole regex in a visualization, please refer to the following one, which is taken from <https://jex.im/regulex>:

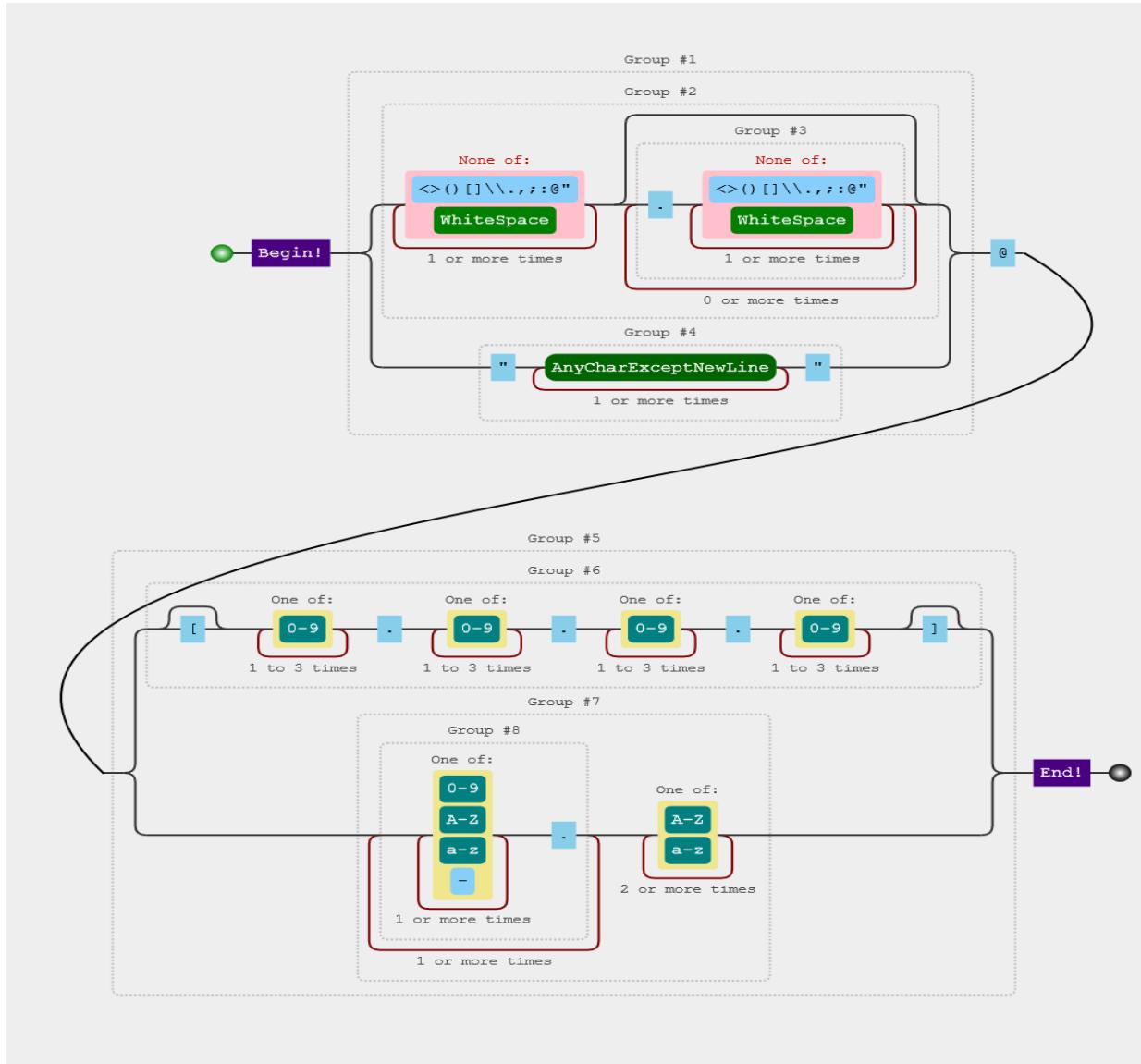


Figure 5.14 – A visualization of the whole email regex

Now, let's proceed with the validation of another important type of information, which is very often subject to typing errors: **dates**.

Checking the validity of dates

Even in the case of dates, the clueless regex developer might think that the following regex is enough to validate dates (in the format of dd-mm-yyyy): `^\d{1,2}([-\/])\d{1,2}\1(?:\d{4}|\d{2})$`. There are two new expressions, never before encountered, that are worth exploring:

- `\1`: This is the **backreference** to group 1. As we explained in the previous section, parentheses also help capture a portion of the string, which you can reference during the rest of the regex. In this case, the `\1` syntax indicates that, at exactly that position, you can expect the same portion of the string matched by the first pair of parentheses. Bear in mind that, in Python, you need to use the `\g<1>` syntax instead of `\1`.
- `(?: ...)`: This is the so-called **non-capturing group**. Sometimes, you need parentheses to correctly apply a quantifier, but you don't want their contents to be reported in the results.

Translating the whole regex into spoken language gives us the following:

1. `^` : This matches the beginning of the string or a line if the multiline flag is enabled.
2. `\d{1,2}` : This matches any digit between 1 and 2 repetitions.
3. `([\-\\/])` : This matches any character between `-` and `/`, and captures the result as group 1.
4. `\d{1,2}` : This matches any digit between 1 and 2 repetitions.
5. `\1` : This backreferences to the captured group 1. So, it expects any character between `-` and `/`.
6. `.+` : This matches any character one or more times, except the line break.
7. `(?:\d{4}|\d{2})` : This matches one of the following two alternatives: any digit for exactly 4 repetitions and any digit for exactly 2 repetitions.

You can visualize the whole regex as follows:

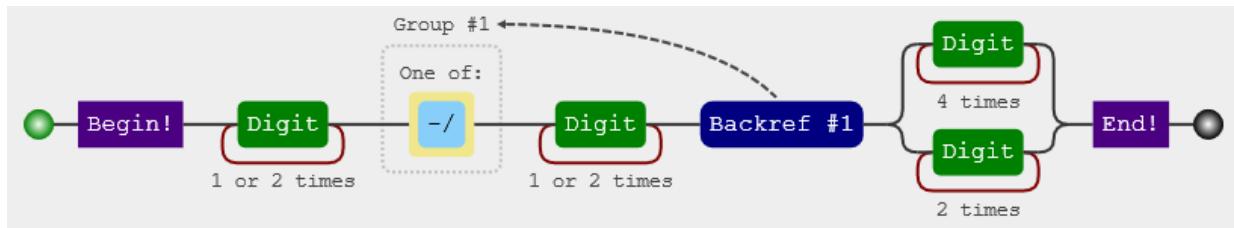


Figure 5.15 – A visualization of the first attempt of a regex for validating dates

As you might have guessed, this regex validates dates in the following formats: `dd-mm-yyyy` , `dd-mm-yy` , `d-mm-yyyy` , and `d-mm-yy` , using `/` or `-` as separators. However, it does not account for errors due to invalid dates, such as February 30 or September 31.

If you want a regex that also accounts for these errors, then you must use the following:

```
^(?:(?:31[\-\\/](?:0?[13578])|(1[02]))[\-\\/](19|20)?\d\d)|(?:(?:29|30)[\-\\/](?:0?[13-9])|(?:(?:0[1-9]|1[0-2])[0-9]{2})[\-\\/](?:0[1-9]|1[0-2])[0-9]{2})
```

Again, viewed in this way, this regex is difficult to interpret. However, looking at it "from above" a bit more, you realize that it consists of four alternatives:

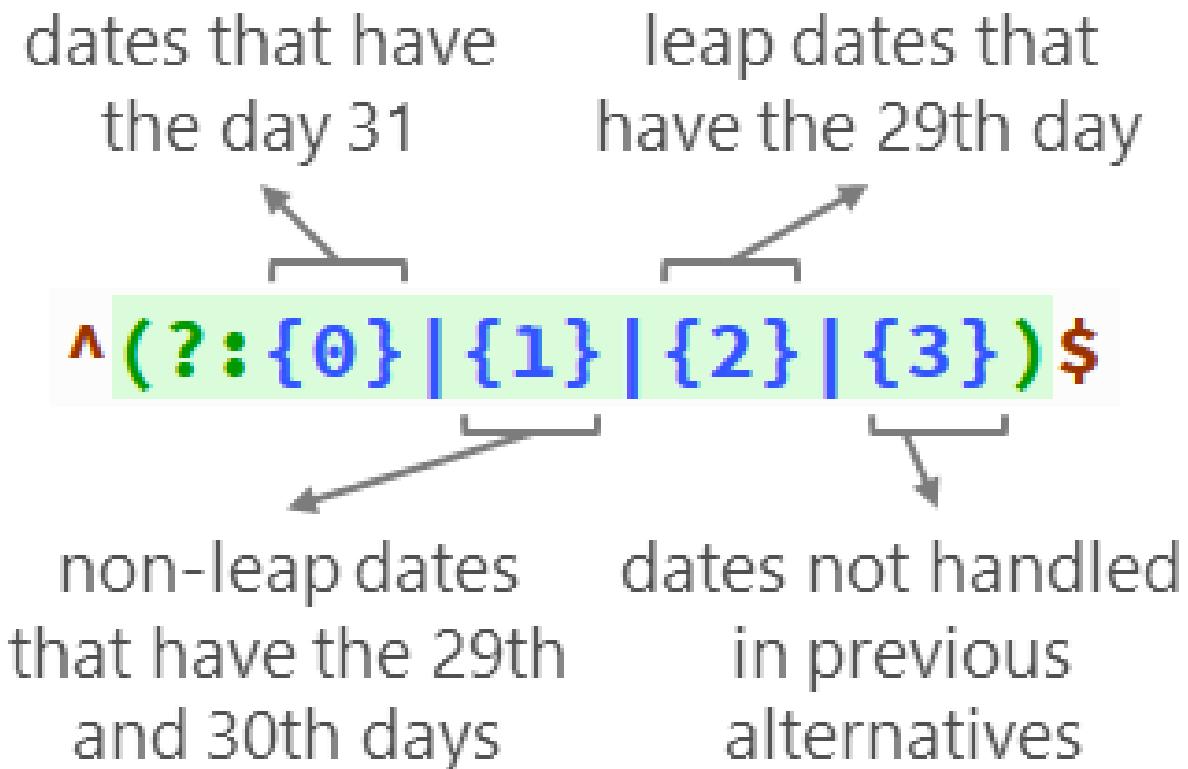


Figure 5.16 – The structure of the complex regex for date validation

Also, in this case, the `{0}`, `{1}`, `{2}`, and `{3}` strings are just placeholders, not characters to match. That said, let's start by defining the `{0}` token.

The `{0}` token matches the dates that have the 31st day. As in previous cases, it's much easier to explain what this regex does with a visualization rather than with words. In *Figure 5.17*, the labels explain every single detail of the subparts:

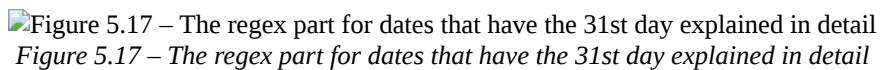


Figure 5.17 – The regex part for dates that have the 31st day explained in detail

The regexes used for the other placeholders are very similar to the one we just explained. Therefore, we will leave the explanation of the others as an exercise for the reader. If you want to see the whole regex in a visualization, please refer to the following diagram, which is taken from <https://jex.im/regulex>:

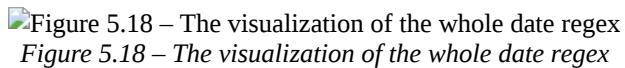


Figure 5.18 – The visualization of the whole date regex

The regex that we just examined allows the validation of the `dd-mm-yyyy` format with all of its variants. In the code, we will demonstrate how you can implement date validation in Power BI. Additionally, you will find regexes that allow you to validate dates in the `mm-dd-yyyy` and `yyyy-mm-dd` formats with all of their variants (where the year is made of two digits, the month is made of one digit, and so on).

Now that you understand what's behind the complex regexes presented earlier, let's move on to implementing them in Power BI to validate your data.

Validating data using regex in Power BI

To date, Power BI has no native feature in Power Query to perform operations via regexes. There are cases when you can't avoid using regexes to extract useful information from data in text form. The only way to be able to use regexes is through R scripts or Python scripts. The only cons you have in this case is that, if you need to publish the report on the Power BI service, to allow Power Query to use external R or Python engines, you must also install the on-premises data gateway in personal mode.

However, let's get right into it with real-world examples.

Let's suppose you work at a retail company where there is a team dedicated to identifying fraudulent customers. As soon as a team member identifies a fraudster, they fill out an Excel spreadsheet, in which the *Email* and *BannedDate* columns are included along with others. Your task is to load the data from this Excel file into Power BI and, from other data sources, select only the fraudster's information in order to carry out specific analysis on their purchases.

Having the correct fraudster emails within the Excel file is critically important to be able to properly join with the other data. Having the correct ban dates is also important in order to know whether further orders from that fraudster have slipped through the cracks after that date. As you know, the filling in of an Excel file by several users is done without any kind of validation of the entered data; therefore, it is subject to human errors. So, identifying any errors when filling out certain fields and highlighting them allows the fraud team to be able to correct them. It is precisely in this case that regexes come to your aid.

Using regex in Power BI to validate emails with Python

In the repository that comes with this book, you can find the `Users.xlsx` Excel file inside the `Chapter05` folder. Its content is similar to *Figure 5.15*:


Figure 5.19 – The content of the Users.xlsx file
Figure 5.19 – The content of the Users.xlsx file

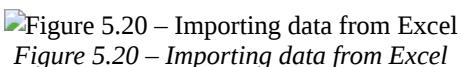
In this section, we will focus exclusively on the `Email` column. This contains the email addresses of fraudsters entered manually by the fraud team, which was described at the beginning of the section. These email addresses are not all syntactically correct. Moreover, in the Excel file, there is also the `IsEmailValidByDefinition` column, whose values (`1=yes; 0=no`) indicate whether the email in correspondence of that value is actually valid or not.

Python has a built-in package, called `re`, which contains all the functions you need to work with regexes. Additionally, in `pandas`, there are several methods for a series or dataframe object, which accept regexes to find a pattern in a string. These methods work in the same way as the ones you will find in Python's `re` module. We will be using the `match` method shortly.

You will learn about the use of the `r'...'` syntax to create strings. This is a **raw string** that allows you to treat the backslash (`\`) as a literal character and not as an escape character.

So, open your Power BI Desktop, make sure the Python environment you use is `pbi_powerquery_env`, and let's get started:

1. From the ribbon, click on the **Excel** icon to import data from Excel:


Figure 5.20 – Importing data from Excel
Figure 5.20 – Importing data from Excel

2. From the **Open** dialog box, select the previously mentioned `Users.xlsx` file.
3. From the **Navigator** window, select the **Users** sheet and then click on **Transform Data**:

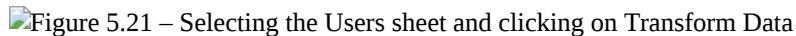

Figure 5.21 – Selecting the Users sheet and clicking on Transform Data

Figure 5.21 – Selecting the Users sheet and clicking on Transform Data

4. Click on the **Transform** menu, and then click on **Run Python Script**.
5. Then, copy and paste the following code into the Python script editor and click on **OK**:

```
import pandas as pd
import re
df = dataset
regex_local_part = r'(^<>()[\]\.,;:\s@"]+(<>()[\]\.,;:\s@"]+)*|(\".+\"")'
regex_domain_name = r'(([a-zA-Z]-[0-9]+.)+[a-zA-Z]{2,})'
regex_domain_ip_address = r'(^([?][0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])?'
pattern = r'^({0})@({1}|{2})$'.format(regex_local_part, regex_domain_name, regex_domain_ip_address)
df['isEmailValidFromRegex'] = df['Email'].str.match(pattern).astype(int)
```

You can also find this Python script in the `01-validate-emails-with-regex-with-python.py` file, which is inside the repository that comes with the book, in the `Chapter05\validating-data-using-regex` folder.

6. Power BI Desktop might display an alert that says the following: **Information is required about data privacy**. If so, click on **Continue**, and follow Step 2; otherwise, you can jump to Step 3.
7. The **Privacy levels** window pops up. Here, you'll specify an isolation level that defines the degree to which one data source will be isolated from other data sources. You could select to *ignore Privacy Levels checks*, but this might expose confidential data to unauthorized persons. You will be asked to select a privacy level for both the Python script and the dataset loaded from Excel. If you select the **Organizational** level for both, everything works fine on Power BI Desktop. However, if you plan to publish your reports to the *Power BI service (or Embedded)*, you *must use the "Public" level*. For more details, please refer to <http://bit.ly/pbi-privacy-levels>. For now, select the **Organizational** level for both options.
8. For now, we are only interested in the `df` dataset. So, click on its **Table** value:

Figure 5.22 – Selecting the df dataset as a result of the Python script transformation

Figure 5.22 – Selecting the df dataset as a result of the Python script transformation

9. As you can see, the **isEmailValidFromRegex** column is added and it contains the Boolean values resulting from the validation of the emails through your regex. If you do a check, you will see that they coincide with the values given by definition in the **IsEmailValidByDefinition** column:

Figure 5.23 – The regex validation results for emails

Figure 5.23 – The regex validation results for emails

Your regex did a great job! Now you can go back to the **Home** menu and click on **Close & Apply**.

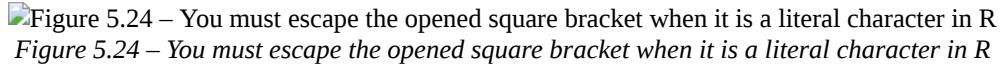
Thanks to the **isEmailValidFromRegex** column, you can now appropriately filter the correct and incorrect email addresses, perhaps even reporting the matter to the fraud team in a dedicated report.

Using regex in Power BI to validate emails with R

If you want to use R for your email validation using regex, the process is pretty much the same except for a few things.

First of all, *R only allows the use of raw strings as of version 4.0.0*. Additionally, the syntax for raw strings is slightly different. Instead of `r'...'`, you can use `r'(...)'`, `r'[...]'`, or `r'{...}'` indiscriminately. Additionally, instead of using numeric placeholders in curly brackets inside the string and then assigning them via the `format()` function, as you would in Python, in R, you can simply use the variable names in curly brackets directly as placeholders.

That said, the second thing you need to pay attention to is the following: you have to be careful so that, in R, not only `[` is considered a metacharacter, but also `[`. Therefore, when you want to use both square brackets as literal characters, you must prepend the backslash escape character (`\`) for both. Therefore, the part of the regex that identifies the character class in the local-part regex of the email is slightly different:



R:Base provides two functions that enable you to use regexes: `grep()` and `grep1()`:

- `grep1()` returns a Boolean value based on whether a pattern exists in a character string.
- `grep()` returns the indexes of the occurrences in the vector of characters that contains a match or the specific strings that have the match.

Since we want to adopt the *Tidyverse paradigm*, we will use the wrapper functions provided by the **stringr** package, which are `str_detect()` and `str_which()`, respectively.

Having clarified these differences, the process to validate the emails present in the `Users.xlsx` Excel file in Power BI using the R script is practically the same as that discussed in the previous section where we used Python:

1. Repeat *Steps 1 to 3* of the previous section to import the data contained in the `Users.xlsx` file.
2. Click on the **Transform** menu, and then click on **Run R Script**.
3. Then, copy and paste the following code into the R script editor and click on **OK**:

```
library(dplyr)
library(stringr)
regex_local_part <- r'(([<>()\\[\\]\\.,;:\\s@\\"]+\\.\\.\\s@\\"]+)*|(\\\".+\\")'
regex_domain_name <- r'(([a-zA-Z-0-9]+.)+[a-zA-Z]{2,})'
regex_domain_ip_address <- r'((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\]?)'
pattern <- str_glue(
  '^({regex_local_part})@({regex_domain_name}|{regex_domain_ip_address})$'
)
df <- df %>%
  mutate( isEmailValidFromRegex = as.integer(str_detect(Email, pattern)) )
```

You also can find this R script in the `02-validate-emails-with-regex-with-r.R` file, which is the repository that comes with the book, in the `Chapter05\validating-data-using-regex` folder.

4. Power BI Desktop might display a notice that says the following: **Information is required about data privacy**. If so, click on **Continue**, and follow the instructions here. Otherwise, you can jump to *Step 5*. Also, select the *Organizational* level for R scripts. Sometimes, you might find that the compatibility levels of datasets and analytical scripts are not compatible with each other. In this case, Power BI might give you an alert, such as **Formula.Firewall: Query 'XXX' (step 'YYY') is accessing data sources that have privacy levels which cannot be used together. Please rebuild this data combination**. In this case, simply open the **Data source settings** window, as follows:

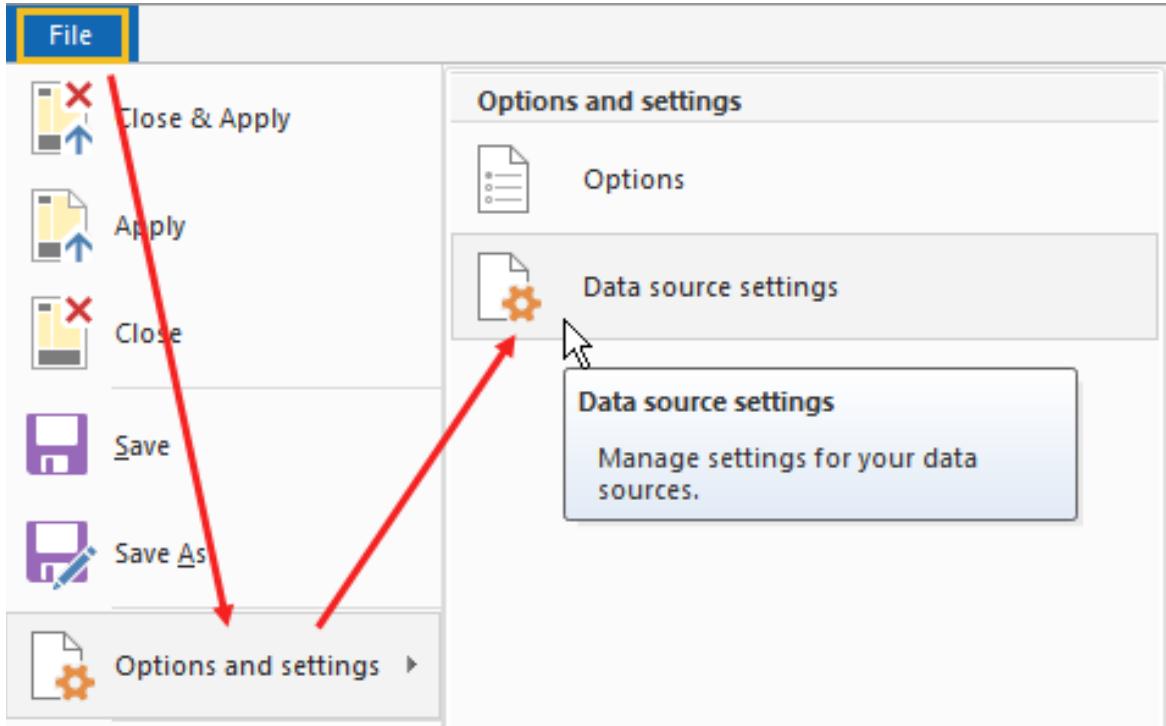


Figure 5.25 – Opening the Power Query Data source settings window

After that, you have to make sure that all data sources have the same level of privacy (in our case, *Organizational*), changing it for each of them, if necessary, through the **Edit Permissions...** option:



Figure 5.26 – Editing the privacy permissions for the data sources

At this point, you can refresh the preview data by clicking on **Refresh Preview**.

5. Additionally, in this case, we are only interested in the `df` dataset. So, click on its **Table** value:

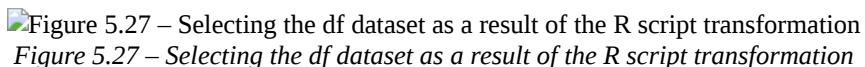


Figure 5.27 – Selecting the df dataset as a result of the R script transformation

6. As you can see, the `isEmailValidFromRegex` column has been added, and it contains the Boolean values (transformed to 1 and 0) resulting from the validation of the emails through your regex. If you do a check, they coincide with the values given by definition in the `isEmailValidByDefinition` column. Your regex did a great job! Now you can go back to the **Home** menu and click on **Close & Apply**.

Thanks to the `isEmailValidFromRegex` column, you can now appropriately filter the correct and incorrect email addresses in your reports.

Now, let's take a look at how to validate dates in Power BI with Python.

Using regex in Power BI to validate dates with Python

As an example of the dates to validate, we will use the `Users.xlsx` Excel file that we used earlier. It contains the `BannedDate` column that has string values representing dates in the `mm/dd/yyyy` format with all its variants.

Moreover, in the Excel file, there is also the *IsDateValidByDefinition* column, whose values (1=yes; 0=no) indicate whether the date matching that value is valid or not.

By now, you already know the Python functions needed to use a regex. So, let's get started:

1. Repeat Steps 1 to 3 of the *Using regex in Power BI to validate emails with Python* section to import the data contained in the `users.xlsx` file.
2. Click on the **Transform** menu and then click on **Run Python Script**.
3. Then, copy and paste the following code into the Python script editor and click on **OK**:

```
import pandas as pd
import re
df = dataset
regex_dates_having_day_31 = r'^(?:(?:0?[13578])|(?1[02]))[-\v]31[-\v](?:19|20)?\d\d'
    regex_non_leap_dates_having_days_29_30 = r'^(?:(?:0?[13-9])|(?1[0-2]))[-\v](?:29|30)[\v'
    regex_leap_dates_having_day_29 = r'^(?:0?2[-\v]29[-\v](?:19|20)?(?:02468)[048])|(?:
    regex_remaining_dates = r'^(?:0?(?:0?[1-9])|(?1[0-2]))[-\v](?:0?1\d)|(?0?[1-9])|(?:
pattern = r'^({0}|{1}|{2}|{3})$'.format(regex_dates_having_day_31, regex_non_leap_dates_having_
df['isValidDateFromRegex'] = df['BannedDate'].str.match(pattern).astype(int)
```

You can find a more exhaustive Python script in the `03-validate-dates-with-regex-with-python.py` file, which can be found in the repository that comes with the book, in the `Chapter05\validating-data-using-regex` folder. That script handles dates in the formats of `mm-dd-yyyy`, `dd-mm-yyyy`, and `yyyy-mm-dd` with all their variances, including both `-` and `/` as separators.

4. If Power BI requires you to provide it with data privacy information, you already know how to proceed based on what we've discussed in the previous sections.
5. As you can see, the `isValidDateFromRegex` column has been added, and it contains the Boolean values resulting from the validation of the emails through your regex. If you do a check, they coincide with the values given by definition in the `IsDateValidByDefinition` column:

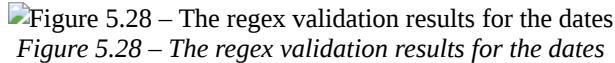


Figure 5.28 – The regex validation results for the dates

Figure 5.28 – The regex validation results for the dates

Your regex did a great job! Now you can go back to the **Home** menu and click on **Close & Apply**.

Thanks to the `isValidDateFromRegex` column, you can now filter the correct and incorrect email addresses and work appropriately with them.

Using regex in Power BI to validate dates with R

If you want to use R for your date validation using regex, in this case, the process is pretty much the same except for what you have already learned in the *Using regex in Power BI to validate emails with R* section. Starting from the same `Users.xlsx` Excel file that we used in the previous section, here are the steps to follow:

1. Repeat Steps 1 to 3 of the *Using regex in Power BI to validate emails with Python* section to import the data contained in the `users.xlsx` file.
2. Click on the **Transform** menu, and then click on **Run R Script**.
3. Then, copy and paste the following code into the R script editor and click on **OK**:

```
library(dplyr)
library(stringr)
df <- dataset
regex_dates_having_day_31 <- r'((?:0?[13578])|(?1[02]))[-\v]31[-\v](?:19|20)?\d\d'
regex_non_leap_dates_having_days_29_30 <- r'((?:0?[13-9])|(?1[0-2]))[-\v](?:29|30)[\v
regex_leap_dates_having_day_29 <- r'((?:0?2[-\v]29[-\v](?:19|20)?(?:02468)[048])|(?:
regex_remaining_dates <- r'((?:0?(?:0?[1-9])|(?1[0-2]))[-\v](?:0?1\d)|(?0?[1-9])|(?2[0
```

```

pattern <- str_glue(
  '^(:{regex_dates_having_day_31}|{regex_non_leap_dates_having_days_29_30}|{regex_leap_date})'
)
df <- df %>%
  mutate( isDateValidFromRegex = as.integer(str_detect(BannedDate, pattern)) )

```

You can find a more exhaustive R script in the `04-validate-dates-with-regex-with-r.R` file, which can be found in the repository that comes with the book, in the `Chapter05\validating-data-using-regex` folder. That script handles dates in the formats of `mm-dd-yyyy`, `dd-mm-yyyy`, and `yyyy-mm-dd` with all their variances, including both `-` and `/` as separators.

4. If Power BI requires you to provide it with data privacy information, you already know how to proceed based on what we've discussed in the previous sections.
5. As you can see, the **isValidDateFromRegex** column has been added, and it contains the Boolean values resulting from the validation of the emails through your regex. If you do a check, they coincide with the values given by definition in the **IsDateValidByDefinition** column. Your regex did a great job! Now you can go back to the **Home** menu and click on **Close & Apply**.

Thanks to the **isValidDateFromRegex** column, you can now appropriately filter the correct and incorrect dates in your reports.

In the next section, you will learn how to import the contents of a semi-structured log file using Python and R.

Loading complex log files using regex in Power BI

Log files are a very useful tool for developers and administrators of computer systems. They record what happened to the system, when it happened, and which user actually generated the event. Thanks to these files, you can find information about any system failure, thus allowing a faster diagnosis of the causes of these faults.

Logs are often **semi-structured data**, that is, information that cannot be persisted in a relational database in the format in which it is generated. In order to be analyzed with the usual tools, first, this data must be transformed into a more suitable format.

Since they are not structured data, it is difficult for them to be imported into Power BI as is, unless someone has developed a custom connector to do so. It is in these scenarios that using a regex in languages such as Python or R can help us get the desired results.

Apache access logs

Let's suppose your company has a website published through an Apache web server. Your manager asks you to carry out an analysis regarding which web pages of the site are the most clicked on. The only way to get this information is to analyze the *access log file*. This file records data about all requests made to the web server. Here is an example of an Apache access log:

Figure 5.29 – An example of an Apache access log
Figure 5.29 – An example of an Apache access log

As you can see, at first glance, there is a fairly organized structure to the information in this log. If no one has customized the output of the Apache log files, it uses the **Common Log Format (CLF)** by default. You can find a real example of an Apache access log in the `apache_logs.txt` file, which is inside the repository that comes with this book, in the `Chapter05\loading-complex-log-files-using-regex` folder. We found it in the GitHub repository at <http://bit.ly/apache-access-log> (click on **Download** to view it).

If you go ahead and read the documentation of those log files, you will deduce that the information recorded in the access log follows the *NCSA extended/combined log format*. So, the data that is recorded is as follows:

1. The remote hostname (the IP address).

2. The remote logname (if empty, you'll find a dash; it is not used in the sample file).
3. The remote user if the request was authenticated (if empty, you'll find a dash).
4. The datetime that the request was received, in the [18/Sep/2011:19:18:28 -0400] format.
5. The first line of the request made to the server between double quotes.
6. The HTTP status code for the request.
7. The size of the response in bytes, excluding the HTTP headers (could be a dash).
8. The Referer HTTP request header, which contains the absolute or partial address of the page making the request.
9. The User-Agent HTTP request header, which contains a string that identifies the application, operating system, vendor, and/or version of the requesting user agent.

Once you know both the nature of the information written in the log and the form in which it is written, you can take advantage of the powerful tools provided by regexes to better structure this information and import it into Power BI.

Importing Apache access logs in Power BI with Python

As mentioned earlier, you can find a real example of an Apache access log in the apache_logs.txt file, which is inside the repository that comes with this book, in the Chapter05\loading-complex-log-files-using-regex folder. You will load the information in this file using a Python script, not a Power BI connector.

Compared to what you've learned before about regexes and Python, in the 01-apache-access-log-parser-python.py Python script (which you'll find in the preceding folder), you'll encounter these new constructs:

- To be able to read a text file line by line in Python, you'll use the `open(file, mode)` functions and the `readlines()` method. Specifically, you're going to read the apache_logs.txt file as read-only ('r') and read each of its lines to store them in a list.

In regexes, it is possible to refer to groups identified by round brackets not only by a numerical index but also by a name. This is thanks to **named capturing groups**. Usually, the regex syntax that is used to assign a name to a group is `(?<group-name>...)`. In Python, it is `(?P<group-name>...)`:

- In Python, you can define a list of regex parts that can be merged together (`join`) using a separator, which is defined by a regex itself (`\s+`):

```
regex_parts = [
    r'(?P<hostName>\S+)',
    r'\S+',
    r'(?P<userName>\S+)',
    r'\[(?P<requestDate>[\w:/]+)\s[\+\-\d{4}]\]',
    r'"(?P<requestContent>\S+\s?\S+?\s?\S+?"',
    r'(?P<requestStatus>\d{3}|-)',
    r'(?P<responseSizeBytes>\d+|-)',
    r'"(?P<requestReferrer>[^"]*)"',
    r'"(?P<requestAgent>[^"]*)?"',
]
pattern = re.compile(r'\s+'.join(regex_parts) + r'$')
```

Note that, in this case, the `re.compile()` function is used since the match must be done many times on all lines of the log; therefore, precompiling the regex could have computational advantages.

- Pattern matching is done for each line in the log:

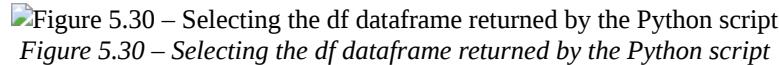
```
for line in access_log_lines:
    log_data.append(pattern.match(line).groupdict())
```

The `groupdict()` method returns a dictionary with the group names as the key and the matched strings as the value for that key. All the dictionaries for each line are appended to the `log_data` list.

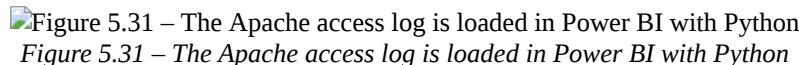
We leave it to the reader to interpret how each individual regex part goes about capturing the desired string.

Now that we've clarified a few points in the code, let's import the log into Power BI:

1. In Power BI Desktop, be sure to use the `pbi_powerquery_env` environment.
2. Go to **Get Data** and select the Python script.
3. Copy and paste the script from the `01-apache-access-log-parser-python.py` file into the Python script editor and click on **OK**.
4. Then, select the **df** dataframe from the **Navigator** window and click on **Load**:



5. If you click on the **Data** icon, you can view the entire log loaded as a structured table:



Awesome! Thanks to the power of regex, you've just managed to easily import into Power BI what looked like a complex log file to manage.

Importing Apache access logs in Power BI with R

In this section, you will load the information of the `apache_logs.txt` file, but this time, using an R script.

Compared to what you've learned previously about regexes in R, in the `02-apache-access-log-parser-r.R` script (which you'll find in the same preceding folder), you'll encounter these new constructs:

- To be able to read a text file line by line in R, you'll use the `read_lines()` function from the `readr` package. Specifically, you're going to read each line of the `apache_logs.txt` file in order to persist them to a vector.
- In order to take full advantage of named capturing groups in R, you need to install and use the features of a new package, called **namedCapture**. Thanks to this package, both regex syntaxes for named groups are allowed: the standard `(?<group-name>...)` regex syntax and the `(?P<group-name>...)` regex syntax.
- Just as we did in the Python script, in R, you'll also define a vector of regex parts, which you'll merge with the `paste(..., collapse = '...')` function. This function has the task of joining regex parts together through the `\s+` separator. After merging all of the parts, the `$` character is added to the end of the resulting string using the `paste0(...)` function. Remember that raw strings have a different syntax in R than in Python. In this case, we will use the `r'{...}'` syntax:

```
regex_parts <- c(  
  r'(?P<hostName>\S+)'  
, r'\S+'  
, r'(?P<userName>\S+)'  
, r'\[(?P<requestDateTime>[\w:/]+\s[+-]\d{4})\]'  
, r'"(?P<requestContent>\S+\s?\S+\s?\S+?)"'  
, r'(?P<requestStatus>\d{3}|-)'  
, r'(?P<responseSizeBytes>\d+|-)'  
, r'"(?P<requestReferrer>[^"]")'  
, r'"(?P<requestAgent>[^"]*)?"'  
)  
pattern <- paste0( paste(regex_parts, collapse = r'{\s+}'), '$' )
```

- Pattern matching is done using the `str_match_named()` function of the `namedCapture` package over the whole log vector, using a single-line command:

```
df <- as.data.frame( str_match_named( access_log_lines, pattern = pattern ) )
```

Again, we leave it to the reader to interpret how each individual regex part goes about capturing the desired string.

Now that we've clarified a few points in the code, let's import the log into Power BI:

1. First, you need to install the `namedCapture` package. So, open RStudio and make sure that the engine being referenced is the latest one in **Global Options**. Then, run the following code in a new script to temporarily set CRAN as the repository to download packages from:

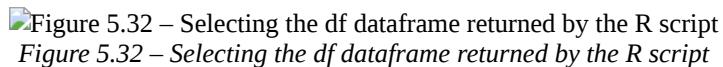
```
local({  
  r <- getOption("repos")  
  r["CRAN"] <- "https://cloud.r-project.org/"  
  options(repos = r)  
})
```

All this has been done to download the latest version of the `namedCapture` package.

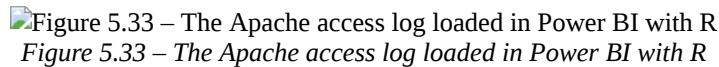
2. Now, go to the console and enter and run the following code:

```
install.packages("namedCapture")
```

3. Open Power BI Desktop, go to **Get Data**, and select the R script.
4. Copy and paste the script from the `02-apache-access-log-parser-r.R` file into the R script editor and then click on **OK**.
5. Then, select the **df** dataframe from the **Navigator** window and click on **Load**:



6. If you click on the **Data** icon, you can view the entire log loaded as a structured table:



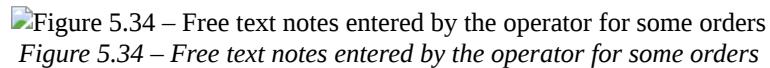
Great job! You were able to import a semi-structured log file into Power BI even with R.

Extracting values from text using regex in Power BI

The last use case we want to present happens very often when dealing with shipments of goods to customers. Sometimes, it happens that a fraudster manages to steal the goods addressed to a customer; therefore, the customer must be refunded by the company. The defrauded customer then contacts Customer Care to request a refund. If the management system provided to the Customer Care operator who has to manage the case does not allow you to enter the information of the refund in a structured way, the operator must resort to the only possible method: the entry of a *free text note* associated with the order, which specifies the *amount*, the *reason* and the *date* of the refund.

You already know that information entered in free text is every analyst's nightmare, especially when your boss asks you to analyze the very information entered in these infamous notes.

In the repository that comes with this book, you can find the `OrderNotes.xlsx` Excel file inside the `Chapter05` folder. Its content is similar to the content shown in *Figure 5.34*:



As you can see, by looking at the contents of the Excel file, the relevant information to extract from the notes is as follows:

- The refund amount
- The refund reason
- The refund date

The problem is that the Customer Care operators used a lot of imagination to enter this information, without the slightest predetermined rule of how to structure it. From this, we can see the following:

- The refund amount was entered as *EUR xx.yy*, *EURxx.yy*, *xx.yy EUR*, *€ xx.yy*, *xx.yy€*, and *xx.yy €*.
- The "separator" between all of the pieces of information can be made by one or more whitespaces or by a dash surrounded by one or more spaces.
- The refund date is always in the *dd/mm/yyyy* format (you are lucky here!).
- The refund reason could contain any text.

Given all this generality of the entered notes, is it possible to correctly extract the information needed for the analysis requested by your boss? The answer is certainly "yes" if you know how to best use regexes.

One regex to rule them all

With the experience you gathered during the previous sections, you will immediately understand the solution we are going to propose. Consider the following regex parts:

- **Currency:** `(?:EUR|€)`
- **Separator:** `(?:\s+)?-?(?:\s+)?`
- **Refund amount:** `\d{1,}\\.?\d{0,2}`
- **Refund reason:** `.*?`
- **Refund date:** `\d{2}[\-\//]\d{2}[\-\//]\d{4}`

Remember the syntax of a **non-capturing group**, `(?:...)`? Well, with this syntax, you're explicitly saying to the regex engine that you don't want to capture the content inside those brackets, as this isn't important information to extract. That said, the final regex is nothing more than multiple alternative combinations of these parts, such as the one you can see in *Figure 5.35*:

Figure 5.35 – The full regex structure for extracting information from notes
Figure 5.35 – The full regex structure for extracting information from notes

If you're curious to see it in full, the final complete regex is as follows:

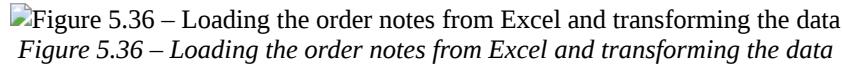
```
^(?:(?:EUR|€)(?:\s+)?-?(?:\s+)?(?:P<RefundAmount>\d{1,}\\.?\d{0,2})(?:\s+)?-?(?:\s+)?(?:P<RefundR
```

Let's implement it in Power BI using Python.

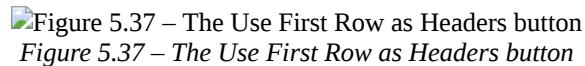
Using regex in Power BI to extract values with Python

As you saw from *Figure 5.35*, our regex contains named groups that are *reused multiple times* within it. Unfortunately, the reuse of the same named group within a regex is not supported by the Python `re` module, which, by the way, is the module that is also used behind the scenes in `pandas`. In order to use more advanced features of regex, such as the previously mentioned *identically named groups* or *lookbehind* and *lookahead* syntaxes (which are not explored in this chapter), you must use the `regex` module. So, first of all, you have to install it within your `pbix_powerquery_env` environment. Then, you have to load the `orderNotes.xlsx` Excel file, which you can find in the `Chapter05` folder, into Power BI Desktop. After that, you can transform that dataset using a Python script. So, let's get started:

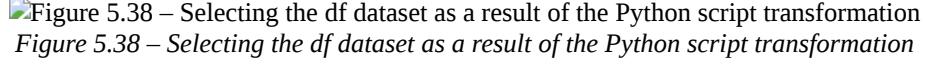
1. Open your Anaconda Prompt, switch to your `pbi_powerquery_env` environment using the `conda activate pbi_powerquery_env` command, and then install the `regex` package using this code: `pip install regex`.
2. Open your Power BI Desktop and make sure the referenced Python environment is `pbi_powerquery_env` in the **Options**.
3. From the ribbon, click on the **Excel** icon to import data from Excel and open the `OrderNotes.xlsx` file.
4. Select the **Sheet1** dataset from the **Navigator** window and click on **Transform Data**:



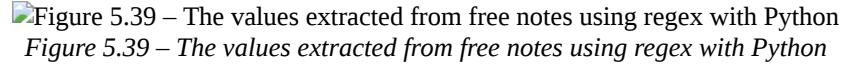
5. Declare the first row of loaded data as column headers by clicking on **Use First Row as Headers**:



6. Then, go to the **Transform** menu and click on **Run Python script**.
7. Open the Python script that you can find inside the `01-order-notes-parser-python.py` file, which is in the `Chapter05\extracting-values-from-text-using-regex` folder. Copy and paste the script into the **Run Python script** editor and then click on **OK**.
8. If there are any issues with the compatibility levels of the datasets, simply open the **Data source settings** window and set the permissions of each dataset to *Organizational* using **Edit Permissions....**. Then, click on **Refresh Preview**.
9. We are only interested in the `df` dataset. So, click on its **Table** value:



10. You can see that the resulting table has three more columns, where each one is related to a named group:

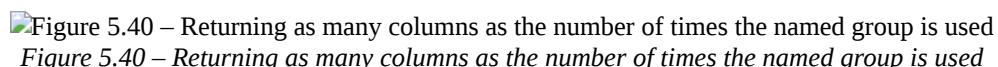


Finally, go to the **Home** menu and click on **Close & Apply**.

Awesome! You've just managed to reorganize the data contained in the order notes using regexes in Python. Your boss will be more than satisfied!

Using regex in Power BI to extract values with R

In R, we can continue to use the `namedCapture` package to manage named groups that are reused multiple times in the same regex. We can do this by putting the `(?J)` modifier in front of it (this allows multiple named capturing groups to share the same name). Unlike Python, in R, the `str_match_named()` function of the `namedCapture` package does not return a one-time result captured by the named group. It returns as many columns as the number of times it was used:



For this reason, we had to further manipulate the result; first, by replacing the empty characters with the null value of `NA`, and second, by applying the `coalesce()` function of `dplyr`, which merges multiple columns into one by

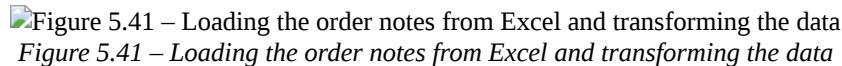
keeping the non-null values.

Important note

We pointed out this limitation to Toby Dylan Hocking, the author of the `namedCapture` package, who recently implemented the feature inside the new version of the package, named `nc`. You can find details of the implementation at <https://github.com/tdhock/namedCapture/issues/15>. The new version of the `nc` package has not yet been published onto CRAN at the time of writing. Therefore, we thought it appropriate to keep the use of the `namedCapture` package in our code. However, feel free to adopt the new `nc` package for your future projects.

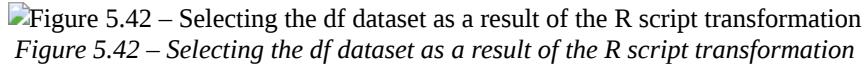
That said, let's start to extract values from the order notes using R in Power BI:

1. Open your Power BI Desktop and make sure the referenced R engine is the latest one (in our case, this is `MRO 4.0.2`).
2. From the ribbon, click on the **Excel** icon to import data from Excel, and open the `OrderNotes.xlsx` file, which you can find in the `Chapter05` folder.
3. Select the **Sheet1** dataset from the **Navigator** window, and click on **Transform Data**:

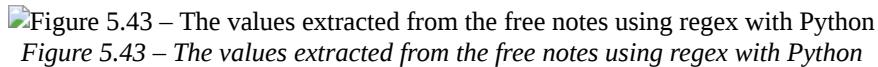

Figure 5.41 – Loading the order notes from Excel and transforming the data

4. Declare the first row of loaded data as column headers by clicking on **Use First Row as Headers**.
5. Then, go to the **Transform** menu and click on **Run R script**.
6. Open the R script that you can find inside the `02-order-notes-parser-r.R` file, which is in the `Chapter05\extracting-values-from-text-using-regex` folder. Copy and paste the script into the **Run R script** editor and then click on **OK**.

7. We are only interested in the `df` dataset. So, click on its **Table** value:


Figure 5.42 – Selecting the df dataset as a result of the R script transformation

8. You can see that the resulting table has three more columns, each one related to a named group:


Figure 5.43 – The values extracted from the free notes using regex with Python

Finally, go to the **Home** menu and click on **Close & Apply**.

Amazing! You've just demonstrated that you know how to rearrange data contained in the order notes using regexes even with R.

Summary

In this chapter, you were introduced to the basics of how to use regexes. Using the bare minimum, you were able to effectively validate strings representing email addresses and dates in Power BI, using both Python and R.

Additionally, you learned how to extract information from semi-structured log files through the use of regexes, and how to import the extracted information, in a structured way, into Power BI.

Finally, you learned how to use regex in Python and R to extract information from seemingly unprocessable free text thanks to the real-world case of notes associated with sales orders.

In the next chapter, you'll learn how to use some de-identification techniques in Power BI to anonymize or pseudonymize datasets that show sensitive data about individuals in plain text before they are imported into Power BI.

References

For additional reading, please refer to the following books and articles:

1. *Regular Expressions: The Complete Tutorial*, Jan Goyvaerts
(<https://www.princeton.edu/~mlovett/reference/Regular-Expressions.pdf>)
2. *Data Privacy Settings In Power BI/Power Query, Part 1: Performance Implications*
(<https://blog.crossjoin.co.uk/2017/05/24/data-privacy-settings-in-power-bipower-query-part-1-performance-implications/>)

6 Anonymizing and Pseudonymizing your Data in Power BI

It happens very often to those who develop a specific software product for a client to want to *repackage* it and sell it to another client who is interested in similar features. However, if you want to show a few screenshots of the software in a demo to the new client, you should avoid showing any data that might be sensitive. Getting in there and trying to mask the data from a copy of the original software database by hand was definitely one of the tasks the poor hapless developer found themselves having to do in the past, maybe even a few days before the demo.

The scenario described does not require data to be shared with a third-party recipient but aims to successfully demo a product to the customer by displaying like-real data. Therefore, there is no concern about a possible brute force attack by professional analysts with the goal of deriving the original data prior to the de-identification operation.

Things definitely change when you need to share an entire dataset with a third-party recipient. The issue has become more sensitive since 2018, especially in Europe, where the need to give more attention to data privacy and **personally identifiable information (PII)** has become imperative for companies to comply with the requirements of the **General Data Protection Regulation (GDPR)**.

The goal of this chapter is to introduce de-identification techniques using **Python** or **R** scripts that can help the **Power BI** developer prevent a person's identity from being linked to the information shown on the report.

In this chapter, you will learn the following:

- De-identifying data
- Anonymizing data in Power BI
- Pseudonymizing data in Power BI

Technical requirements

This chapter requires you to have a working Internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

De-identifying data

PII, also called **personal information** or **personal data**, is any information relating to an identifiable person. There are two types of PII – *direct* and *indirect*. Examples of **direct identifiers** are your name, your address, a picture of you, or an RFID (Radio Frequency Identification) associated with you. **Indirect identifiers**, on the other hand, are all those pieces of information that don't explicitly refer to you as a person, but somehow make it easier to identify you. Examples of indirect identifiers are your license plate number, your bank account number, the link to your profile on a social network, or your place of work.

The practice of **de-identifying** data is to manipulate PPIs so that it is no longer possible to identify the person who generated them.

There are two options for handling direct and indirect personal identifiers – either you decide to destroy them completely, or you decide to keep them separated from the rest of the data, implementing security measures to prevent anyone from re-identifying the data subject. But let's first explore what some of the most common de-identification techniques are.

De-identification techniques

De-identification is a process that is invisible to end users. After doing a careful study with a team of analysts, generally it is the data manager (or the person acting on their behalf) who decides what information should be de-identified. In the next sections, we will discuss the most commonly used de-identification techniques.

Information removal

The simplest form of de-identification is to remove sensitive information from the dataset:

Figure 6.1 – Anonymization, information removal
Figure 6.1 – Anonymization, information removal

It is clear that one of the disadvantages of this simplistic approach could be that the final dataset no longer conforms to the schema expected by the application that must consume it.

Data masking

Data masking hides information that users with specific roles shouldn't see. It could consist of modifying data using word or character substitution. For example, you can replace a value character with a symbol, such as * or . The following is a typical example of data masking:

Figure 6.2 – Anonymization, data masking
Figure 6.2 – Anonymization, data masking

Keep in mind that if the readable domain name in the email address is not public, but belongs to a recognizable legal entity, the data masking applied does not comply with the GDPR rules, as the workplace becomes recognizable.

Some products include **dynamic data masking** solutions out of the box. These mask or block sensitive information to users based on their role, location, and privileges.

Note

For example, **Microsoft SQL Server** and **Azure SQL Database** provide dynamic data masking as a solution to avoid exposing sensitive data to unauthorized users. The data in the database is not changed, as masking rules are applied in the query results.

Data swapping

Data swapping consists of shuffling the values of a column containing sensitive data for the entire dataset. For example, if you have a column containing an individual's date of birth, this can very well be anonymized using the swapping technique with very good results.

Generalization

Generalization consists of replacing point values with other values that indicate a broader category, to which the initial value belongs. For example:

- An *age* of 25 can be transformed in the values >=18 , or *between 18 and 30*.
- A *birth date*, like 04/11/1989 , can be replaced by the *year of birth* 1989 .
- A *postal zip code* can be replaced by a broader *regional zip code*.

Data perturbation

Data perturbation is a technique that replaces original values by adding some random noise or creating synthetic data. This transformation results in a loss of information that can make the data itself useless.

Tokenization

Tokenization is a technique that replaces original sensitive values with a randomly generated alphanumeric value, called a **token**. Completely random tokens offer the highest security, as the content cannot be re-engineered. There is no mathematical algorithm behind the scenes to get the original value back with an inverse transformation. Therefore, the association between the token and original value is generally maintained in a secured database, and tokens are usually generated by specific token servers. Only the token server talks to the token database.

Hashing

Hashing is a technique similar to tokenization, with the difference that the resulting *token*, called a **hash value**, is generated by a mathematical algorithm, has a fixed length, and is almost impossible to transform back to the original value. If you use the same hashing function with the same input value, you'll always have the same hash value as output. Often, additional text called *salt* is added to the input value to make it more complicated for a brute force attack to reverse engineer the hash value.

Encryption

Like hashing, **encryption** uses a mathematical algorithm to transform sensitive data. As opposed to hashing, it is a two-way transformation that needs a decryption key to reverse engineer an encrypted value. Using an encrypted mapping table can boost performance when decrypting data.

Most productivity tools and database systems are now available with end-to-end encryption built-in.

Note

Two examples are the Microsoft SQL Server or Azure SQL databases, which have the *Always Encrypted* feature out of the box. It works by encrypting the data on the client side and hiding the encryption keys from the server. Even database administrators cannot read information stored in an encrypted column without having explicit permission.

Now that you have an idea of the most common transformations used to de-identify sensitive information, you will see how they are used in anonymization and pseudonymization.

Understanding pseudonymization

Pseudonymization is a de-identification process that separates direct or indirect identifiers from the rest of the data, taking care to ensure the following:

- Replacing one or more PII's with **pseudonyms** (a random actual name, but more often a random numeric or alphanumeric identifier), ensuring the non-identification of the subject. Analytical correlations are guaranteed thanks to the fact that the pseudonym is always the same for the same input. So, analysis of pseudonymized data doesn't lose value.
- Not destroying the original PII's, making sure that the entire dataset can be reconstructed (re-identification of the data) using, for example, lookup tables between PII's and pseudonyms, or digital secret keys to pseudonymize inputs into the same output.
- Taking appropriate technical and organizational measures to make it difficult to trace the identity of an individual from the remaining data.

During this process, some de-identification transformations can be made to some PII's that you want to keep in the accessible data. For example, you could replace PII values with similar-looking pseudonyms, making sure to keep track of the replacement in order to guarantee the re-identification.

An example of a pseudonymization process is shown in *Figure 6.2*, where a lookup table is used to guarantee the mapping for the inverse transformation:

Figure 6.3 – The process of pseudonymization
Figure 6.3 – The process of pseudonymization

An architecture of this type also guarantees the possibility of satisfying any requests for deletion of personal data by individuals (as required by GDPR) by meeting the following conditions:

- It will be impossible to identify the subject from that moment on, just by removing the association related to it from the lookup table.
- The complete loss of statistical information useful for data analysis will be avoided, as it's possible to use de-identification transformations that don't erase analytical correlations.

Important Note

Keep in mind that the moment you *permanently lose the link* between a row of accessible data and its respective PPIs, that row becomes *completely anonymized*, thus falling out of GDPR's control.

So, we have introduced the concept of anonymization. Let's take a look at what this means.

What is anonymization?

Anonymization completely destroys direct and indirect identifiers, or destroys the link to their de-identified counterpart, so that there is no danger (or at least it is really very unlikely) that any attacker will be able to reconstruct the identity of the subjects to which the data refers. It has the *non-reversibility of the process* as its main goal. For this reason, the following applies:

Important Note

Anonymized data is outside the scope and control of the GDPR because anonymized data is no longer *personal data*.

The most obvious disadvantage of anonymization is that it removes significant value from the data involved. This is because, after the process is complete, it is impossible to trace the identities that generated that data. It is therefore advisable to assess all relevant risks before anonymizing any dataset.

The second disadvantage of anonymization is that it usually uses randomly generated de-identified strings, so *some statistical information of the dataset is permanently lost*, making any work a data scientist would have to do futile.

It may be that anonymized data can be vulnerable to **de-anonymization attacks**. They consist of enriching the anonymized dataset with available external information, thus imputing the anonymized items. These attacks are more likely to succeed, as the anonymized data is abundant, granular, and fairly stable over time and context.

Usually, the most adopted de-identification techniques for secure anonymization are as follows:

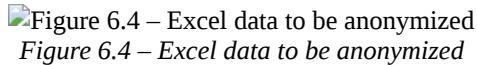
- Tokenization
- Encryption

Let's now see how to apply these concepts to a real case using Power BI.

Anonymizing data in Power BI

One of the possible scenarios that could happen to you during your career as a report developer in Power BI is the following. Imagine you are given an **Excel** dataset to import into Power BI in order to create a report to show to another department of your company. The Excel dataset contains sensitive personal data, such as names and email

addresses of people who have made multiple attempts to pay for an order with a credit card. The following is an example of the contents of the Excel file:

Figure 6.4 – Excel data to be anonymized
Figure 6.4 – Excel data to be anonymized

You are asked to create the report while anonymizing the sensitive data.

The first thing that jumps out at you is that, not only do you have to anonymize the **Name** and **Email** columns, but some names or email addresses can be included in the text of some **Notes**. While locating email addresses is fairly easy using regular expressions, it is not as easy to locate person names in free text. For this purpose, it is necessary to adopt a technique of **natural language processing (NLP)** that goes under the name of **named entity recognition (NER)**. Thanks to NER, it's possible to identify and classify named entities (like people, places, and so on) in free text.

The basic idea is to replace both full names and email addresses with random *tokens*. Depending on the analytical language used, there are different solutions driven by the different packages available that lead to the same result.

Anonymizing data using Python

Python is one of the most widely used languages for performing de-identification transformations in anonymization processes. There are a plethora of packages that implement such solutions. In particular, Microsoft released the open source package **Presidio** (<https://microsoft.github.io/presidio/>), which is to date one of the best solutions for data protection and anonymization. It provides quick identification and anonymization forms for entities found in free text and images, such as credit card numbers, names, locations, social security numbers, email addresses, financial data, and more. **PII recognizers** leverage NER, regular expressions, rule-based logic, and checksums by identifying the relevant context in multiple languages. Behind the scenes, Presidio adopts NLP engines to recognize the entities – it supports both **SpaCy** (the default one) and **Stanza**. One of the most interesting features of Presidio is its *extensibility*. In fact, it is possible to extend the Presidio Analyzer by adding *custom PII entities* very easily.

Once the sensitive entities are identified, you need to replace them with tokens. They are generated in Python using the `secrets` module.

That said, you can find an implementation of this in the Python file, `01-anonymize-data-in-power-bi-python.py`, in the `Chapter06` folder of the GitHub repository that comes with this book. It performs the following operations:

1. Load the libraries needed to execute the code. You will use the `pandas` module, some functions from the `presidio_analyzer` and `presidio_anonymizer` modules, and the `secrets` module.
2. Define two custom functions, one to anonymize emails, the other to anonymize person names. Both use the `analyzer.analyze()` Presidio function to identify the entities to be anonymized and the `secrets` module to generate the tokens into the `anonymizer.anonymize()` function.
3. Initialize the main objects of the **Presidio Analyzer** and **Presidio Anonymizer**.
4. For each row of the dataset previously loaded in Power BI Desktop (or via the `pandas`' `read_excel()` function if you want to test the code in **VSCode**), apply the `anonymizeEmail` function to the `Email` and `Notes` columns and apply the `anonymizeName` function to the `Name` and `Notes` columns. In order to apply a function to each individual value of a column, we adopted the `apply()` function followed by a construct that goes by the name of **lambda function** (introduced by the keyword `lambda`). It is a small function defined without a name (anonymous) to be used inline. Here is an example:

```
df.Name = df.Name.apply(lambda x: anonymizeName(x))
```

In order to proceed, however, it is necessary to configure *a new Python Environment*. This is because, to date, Presidio is supported only for Python versions from 3.6 to 3.8. Your `pbi_powerquery_env` environment has a

newer Python version installed, so you need to create a new environment with Python 3.8. Once created, you have to install the necessary modules to run the code.

These are the steps needed to configure the new environment:

1. Open your **Anaconda Prompt**.

2. Enter and run the following code to create the new `presidio_env` environment with Python 3.8:

```
conda create --name presidio_env python=3.8
```

3. Enter and run the following code to switch to the newly created environment:

```
conda activate presidio_env
```

4. Enter and run the following code to install the Presidio Analyzer:

```
pip install presidio_analyzer
```

5. Enter and run the following code to install the Presidio Anonymizer:

```
pip install presidio_anonymizer
```

6. The Analyzer also installs SpaCy behind the scenes. So, you must also install the SpaCy's *trained pipeline for written English text* (we choose the one for blogs, news, and comments) using this code:

```
python -m spacy download en_core_web_lg
```

This is the largest pipeline used by SpaCy and takes up about 788 MB.

7. Enter and run the following code to install **pandas**:

```
pip install pandas
```

8. If you want to use pandas to directly load the Excel with Python and then test the code before entering it in Power BI, you'll also need the `openpyxl` module:

```
pip install openpyxl
```

9. Enter and run the following code to install `matplotlib`, needed by the Power BI wrapper used with Python scripts:

```
pip install matplotlib
```

You are now ready to apply anonymization in Power BI to the content of the `CustomersCreditCardAttempts.xlsx` Excel file you can find in the `Chapter06` folder.

So, let's get started:

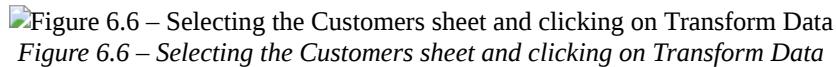
1. Open your Power BI Desktop. Make sure the referenced Python environment is `presidio_env` in the options (its home directory should be `C:\Users\<your-username>\miniconda3\envs\presidio_env`). Keep in mind that in case you can't find the path to a specific environment, activate it in the Anaconda Prompt (`conda activate <your-env>`) and then enter where `python`.

2. From the ribbon, click on the **Excel** icon to import data from Excel:

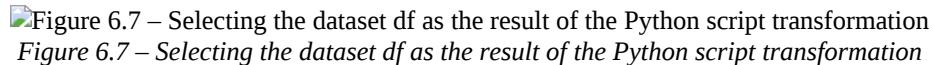


3. From the **Open** dialog box, select the aforementioned `CustomersCreditCardAttempts.xlsx` file.

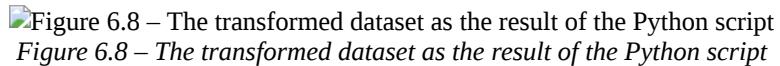
4. From the **Navigator** window, select the `Customers` sheet and then click on **Transform Data**:



5. Click on the **Transform** menu and then click on **Run Python Script**.
6. Copy the script from the `01-anonymize-data-in-power-bi-python.py` file into the Python script editor and click **OK**.
7. If Power BI needs you to provide it with data privacy information, you already know how to proceed based on what you've seen in *Chapter 5, Using Regular Expressions in Power BI*.
8. We are only interested in the `df` dataset. So, click on its **Table** value:



9. As you can see, person names in the `Name` and `Notes` columns and emails in the `Email` and `Notes` columns have been correctly anonymized:



You can now click **Close & Apply** in the **Home** tab.

Notice how the person name contained in the `Notes` column has also been anonymized. This is the result of applying NER algorithms used by the **SpaCy** engine, which works under the hood of Presidio.

Moreover, the de-identification technique used (tokenization) doesn't preserve the statistical characteristics of the dataset, since applying the procedure to the same personal data does not return the same de-identified string.

Note

When you publish the report for which you have anonymized the data, the corresponding dataset will also be published. Users who can only access the dataset (and not the source Excel file) *will only see the anonymized data*, without having the ability to learn about the associations with the original data that has been replaced.

Thanks to anonymization, you can now develop your reports without having to worry about the danger of exposing sensitive data.

Let's see how it is possible to do the same thing in R.

Anonymizing data using R

You can implement a data anonymization process in R as well. As long as you are identifying strings using regular expressions in R, the processing is quite fast. However, when natural language processing techniques like NER need to be implemented, the most widely adopted R packages available often consist of wrappers of open source modules developed in other languages. For example, the **openNLP** R package is an interface to the **Apache OpenNLP** toolkit, based on machine learning algorithms written in **Java**. In order for the `openNLP` package to interface with **OpenNLP** software, its installation also requires as a dependency the `rJava` package, which enables the dialogue between the R and Java worlds.

In order to implement the same anonymization features developed in Python in the previous section in R, you will make use of another widely used R package for NLP operations called `spacyr`. This library provides a convenient R wrapper around the Python `spacy` module. In the previous section, you saw that the Python module called `presidio` installs behind the scenes the same `spacy` module used by `spacyr`. If you're wondering how to run

Python code from an R module, remember that, in *Chapter 3, Configuring Python with Power BI*, you ran Python code through **RStudio** using the R package called `reticulate`. Just as `rJava` takes care of interfacing R with the Java VM, so `reticulate` allows R to interface with a Python environment and execute Python code. In a nutshell, the R code you are going to develop does nothing more than execute the functionality of the `spacy` Python module you used in the previous section.

Note

Remember that you could have used regular expressions to replace email addresses with dummy data. Instead, replacing person names within free text is only possible with an NLP function that recognizes named entities. Hence the need to use a package like `spacyr`.

Just as you did in the previous section, here you will also anonymize the content of the Excel file, `CustomersCreditCardAttempts.xlsx`, through tokenization. Tokens in R will be generated using the `stringi` package.

That said, the R code in the `02-anonymize-data-in-power-bi-r.R` file you can find in the `Chapter06` folder performs the following operations:

1. Load the libraries needed to execute the code. In particular, you will use `stringr`, `dplyr`, and `purrr` from the `Tidyverse` to operate data wrangling; `spacyr` and `stringi` are used for data anonymization.
2. Define the `anonymizeEmails` function, used to anonymize emails, to a free text. It uses the `spacyr` function's `spacy_parse()` with the additional `like_email` attribute. As it can identify multiple email addresses into a single text, the `str_replace_all()` function is used to replace all the found occurrences with a token generated by the `stri_rand_strings` function of the `stringi` package.
3. Define the `anonymizeNames` function, used to anonymize person names, to a free text. It contains more complex logic than the previous function because a person's name can consist of multiple tokens that are not always separated by a space (for example, the name `Roma Rodriguez-Bailey`). Therefore, in order to identify the set of all tokens referring to a single person, we must construct a regex that references the first and last tokens (from the previous example, `Roma.*?Bailey`), which is able to match the entire name. As you can see, there was no need to have to implement all of this logic in the previous section, because the Python `Presidio` module takes care of all of these cases.
4. You must initialize `spacyr` so that it references a Python environment containing the `spacy` module installed. Generally, if you haven't already installed `spacy` in an environment, you can use `spacyr`'s `spacy_install()` function, which sets up a new Python environment with everything you need to make it work properly. In our case, we've already created the Python environment, `presidio_env`, in the previous section, which contains both the `spacy` module and the trained `en_core_web_lg` model to extract language attributes using written English samples taken from the web. It is then enough to reference the environment `presidio_env` path in the `spacy_initialize()` function to correctly configure `spacyr`. Here is the code:

```
spacy_initialize(  
    model = "en_core_web_lg",  
    condaenv = r"C:\Users\<your-username>\miniconda3\envs\presidio_env",  
    entity = TRUE  
)
```

If you run it in RStudio, you'll get something similar to the following message if all is working correctly:

```
successfully initialized (spaCy Version: 2.3.0, language model: en_core_web_lg)  
(python options: type = "condaenv", value = "C:\Users\<your-username>\miniconda3\envs\presidio_env")
```

5. For each row of the dataset previously loaded in Power BI (or using the `readxl` package to test the code in RStudio), apply the `anonymizeEmail` function to the `Email` and `Notes` columns, and apply the `anonymizeName` function to the `Name` and `Notes` columns. In order to apply the two functions defined previously to each element of a column, we used the `map()` function of the `purrr` package. More specifically, `map_chr()` returns the outputs in a vector of strings so that it can replace the column content.

Having briefly explained what the R script does, let's get down to business. In order to use the `spacyr` R package, it must be installed in the latest R engine (in our case, MRO 4.0.2). These are the necessary steps:

1. Open **Rstudio**, and be sure to select in the **Global Options** the most recent MRO engine you already installed, following the steps in *Chapter 2, Configuring R with Power BI*.
2. Since MRO by definition downloads new packages from a default **CRAN (Comprehensive R Archive Network)** snapshot back in time, in order to download the latest version of packages in CRAN you need to overwrite the referenced repository by running the following code:

```
local({  
  r <- getOption("repos")  
  r["CRAN"] <- "https://cloud.r-project.org/"  
  options(repos = r)  
})
```

3. Then, install the `spacyr` package by running the following code in the console:

```
install.packages("spacyr")
```

You are now ready to apply anonymization in Power BI to the content of the `CustomersCreditCardAttempts.xlsx` Excel file using R.

So, let's get started:

1. Open your Power BI Desktop, and make sure the referenced R engine is your latest MRO version in the **Global Options**.
2. From the ribbon, click on the **Excel** icon to import data from Excel.
3. From the **Open** dialog box, select the aforementioned `CustomersCreditCardAttempts.xlsx` file.
4. From the **Navigator** window, select the **Customers** sheet and then click on **Transform Data**:

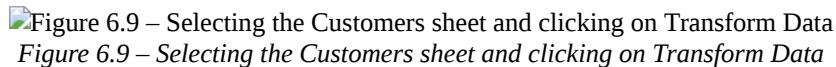


Figure 6.9 – Selecting the Customers sheet and clicking on Transform Data
Figure 6.9 – Selecting the Customers sheet and clicking on Transform Data

5. Click on the **Transform** menu and then click on **Run R Script**.
6. Copy the script from the `02-anonymize-data-in-power-bi-r.R` file into the R script editor and click **OK**. Remember to change the environment path of your machine.
7. If Power BI needs you to provide it with data privacy information, you already know how to proceed based on what you've seen in *Chapter 5, Using Regular Expressions in Power BI*.
8. We are only interested in the `df` dataset. So, click on its **Table** value:

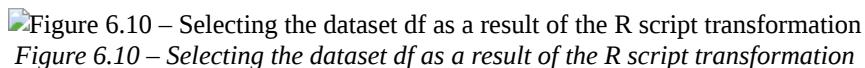


Figure 6.10 – Selecting the dataset df as a result of the R script transformation
Figure 6.10 – Selecting the dataset df as a result of the R script transformation

9. As you can see, person names in the `Name` and `Notes` column and emails in the `Email` and `Notes` columns have been correctly anonymized:

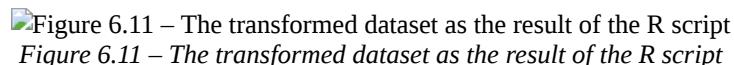


Figure 6.11 – The transformed dataset as the result of the R script
Figure 6.11 – The transformed dataset as the result of the R script

You can then click **Close & Apply** in the **Home** tab.

As you could see, the execution of the R script took longer than that of the Python script. Clearly, the overhead of passing information through `reticulate` makes a difference.

Important Note

If you need to anonymize a not-so-small dataset, it is advisable to do it directly using a Python script for much better performance.

Again, the dataset resulting from the publication of this report will contain only the anonymized data, without giving Power BI users the ability to retrieve the original data.

Let's see now how to pseudonymize the data in Power BI.

Pseudonymizing data in Power BI

Unlike anonymization, pseudonymization maintains the statistical characteristics of the dataset by transforming the same input string into the same output string, and keeps track of replacements that have occurred, allowing those with access to this mapping information to obtain the original dataset again.

Moreover, pseudonymization replaces sensitive data with **fake strings (pseudonyms)**, having the same *form* as the original one, making the de-identified data more realistic.

Depending on the analytical language used, there are different solutions driven by the different packages available that lead to the same result. Let's see how to apply pseudonymization in Power BI to the contents of the same Excel file used in the previous sections with Python.

Pseudonymizing data using Python

The modules and the code structure you will use are quite similar to those already used for anonymization. One difference is that, once the sensitive entities are identified, they are replaced by fake entities of the same type. The fake data generators par excellence in Python are two: **Faker** (<https://faker.readthedocs.io/>) and **Mimesis** (<https://mimesis.readthedocs.io/>). In our example, we'll use Faker, which is inspired by the library of the same name previously developed for **PHP**, **Perl**, and **Ruby**.

Moreover, what changes is the logic of the two custom functions used to de-identify entities and the addition of the management of two dictionaries (`emails_dict` and `names_dict`) to maintain the mapping between personal data and fake data.

We've also added a little more *salt* to the handling of the fake data – person names and email addresses are generated considering the country of each individual in the dataset, passing it as a parameter in custom functions. For example, if the individual is German, the generated person name will be a typical German name.

Let's see in detail what this is all about. The referenced Python file is `03-pseudonymize-data-in-power-bi-python.py`, which you can find in the `Chapter06` folder:

1. While, in the case of anonymization, the `anonymizer.anonymize()` function was used to replace all entities identified using the `analyzer.analyze()` function in one go, now, after the same entity identification, we must *first check if each identified single entity has already been mapped to a fake string*. If the entity is in its own specific dictionary, you retrieve the associated fake string and use that to pseudonymize the text. Otherwise, you generate a new fake string and add it to the dictionary, associating it with the entity in question.
2. When the pseudonymization of all expected columns is complete, the mapping dictionaries (both for names and emails) are persisted to `pk1` files. These are unpickled and used as mapping dictionaries whenever new Excel data needs to be pseudonymized, and then at each refresh of the dataset. This ensures that the same pseudonyms are always used for the same personal data and also for new Excel rows.

In order to use the `faker` module as mentioned before, you need to install it in the `presidio_env` environment in the following way:

1. Open your Anaconda Prompt.

2. Enter and run the following code to switch to the newly created environment:

```
conda activate presidio_env
```

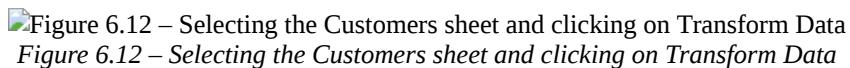
3. Enter and run the following code to install Faker :

```
pip install Faker
```

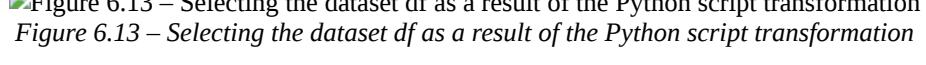
Once this is done, you can start to implement pseudonymization in Power BI:

1. Open your Power BI Desktop and make sure the referenced Python environment is `presidio_env` in **Options** (its home directory should be `C:\Users\<your-username>\miniconda3\envs\presidio_env`).
2. From the Power BI ribbon, click on the **Excel** icon to import data from Excel.
3. From the **Open** dialog box, select the aforementioned `CustomersCreditCardAttempts.xlsx` file.

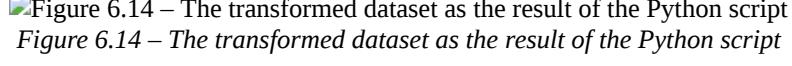
4. From the **Navigator** window, select the **Customers** sheet and then click on **Transform Data**:



5. Click on the **Transform** menu and then click on **Run Python Script**.
6. Copy the script from the `03-pseudonymize-data-in-power-bi-python.py` file into the Python script editor and click **OK**. Remember to change the environment path of your machine.
7. If Power BI needs you to provide it with data privacy information, you already know how to proceed based on what you've seen in *Chapter 5, Using Regular Expressions in Power BI*.
8. We are only interested in the `df` dataset. So, click on its **Table** value:



9. As you can see, person names in the `Name` and `Notes` column and emails in the `Email` and `Notes` columns have been correctly pseudonymized:



You can then click **Close & Apply** in the **Home** tab.

Just as in the case of anonymization, the `Name`, `Email`, and `Notes` columns have been correctly de-identified. What we have in addition is the following:

- The consistency of the person names and emails with the individual's country, although `Faker` does not currently allow you to maintain consistency between generated names and their respective emails. For example, in *Figure 6.14*, you can see an Italian name and an email using a different Italian name at *row 11* (the name *Alfio Migliaccio* is not used in the email).
- The use of the `emails_dict` and `names_dict` mapping dictionaries to ensure that statistical analysis can be done on the pseudonymized dataset.
- The fact that we can trace the original data back thanks to these mapping dictionaries that are persisted to disk.

In this case, when you publish the report to share it with Power BI users, you have the following:

Note

Power BI users who can only access the dataset will only see the de-identified data. By also providing mapping dictionaries to those with the right permissions, you ensure that they can trace the original data back for any legal needs.

Did you notice that, thanks to the Python script, you were able to write to file information resulting from the data processing carried out in Power BI?

Note

Instead of using serialized dictionaries in PKL files, you could have, for example, written the information to CSV or Excel files.

Simply put, you have the ability to log information outside of Power BI. You'll learn more about this possibility in *Chapter 7, Logging Data from Power BI to External Repositories*.

Let's now see how to implement pseudonymization, also in R.

Pseudonymizing data using R

The libraries we will use for data pseudonymization in R are much the same as those used for anonymization in R. To fully simulate the functionality of the Python script developed in the previous section, we also need an R package that generates *fake data* to replace *sensitive data*, such as person names and email addresses. In Python, we used the Faker module, one of the most widely used for that purpose. A package with the same functionality has been developed in R, also inspired by the same code used for Faker, and is called **charlatan** (<https://github.com/ropensci/charlatan>). Additionally, the R code for pseudonymization will follow the same logic already implemented in the Python script in the previous section, with minor differences being as follows:

- Instead of dictionaries, named lists are used for mapping pseudonyms to original entities.
- Named lists are serialized and persisted in **RDS** files, instead of PKL files.

In order to use the `charlatan` R package, it must be installed in the latest R engine (in our case, MRO 4.0.2). These are the necessary steps to follow:

1. Open Rstudio, and be sure to select in the **Global Options** the most recent MRO engine you already installed following the steps in *Chapter 2, Configuring R with Power BI*.
2. Since MRO by definition downloads new packages from a default CRAN snapshot back in time, in order to download the latest version of packages in CRAN you need to overwrite the referenced repository by running the following code:

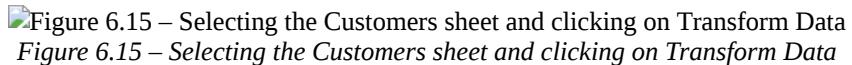
```
local({  
  r <- getOption("repos")  
  r["CRAN"] <- "https://cloud.r-project.org/"  
  options(repos = r)  
})
```

3. Then, install the `charlatan` package by running the following code in the console:

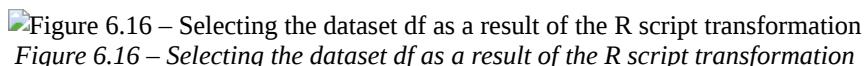
```
install.packages("charlatan")
```

You are now ready to apply pseudonymization in Power BI to the content of the `customersCreditCardAttempts.xlsx` Excel file using R. So, let's get started:

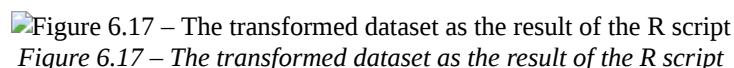
1. Open your Power BI Desktop, and make sure the referenced R engine is your latest MRO version in the **Global Options**.
2. From the ribbon, click on the **Excel** icon to import data from Excel.
3. From the **Open** dialog box, select the aforementioned `customersCreditCardAttempts.xlsx` file.
4. From the **Navigator** window, select the **Customers** sheet and then click on **Transform Data**:



5. Click on the **Transform** menu and then click on **Run R Script**.
6. Copy the script from the `04-pseudonymize-data-in-power-bi-r.R` file into the R script editor and click **OK**. Remember to change the needed paths in the code.
7. If Power BI needs you to provide it with data privacy information, you already know how to proceed based on what you've seen in *Chapter 5, Using Regular Expressions in Power BI*.
8. We are only interested in the `df` dataset. So, click on its **Table** value:



9. As you can see, person names in the `Name` and `Notes` column and emails in the `Email` and `Notes` columns have been correctly anonymized:



You can then click **Close & Apply** in the **Home** tab.

Unfortunately, charlatan does not yet support the `it_IT` locale. But since it is an open source project, it is possible that it will be implemented by the community soon. This, however, does not prevent us from obtaining a very good pseudonymization of the dataset and of the report that will be published on the Power BI service, since, in the absence of a specific locale, the default one (`en_US`) is always used.

It has been said that the implementation of de-identification procedures in R is certainly less performant than in Python. However, the gap can be partially bridged by the introduction of parallelization of operations with multitasking. We'll look at this technique in detail in *Chapter 8, Calling External APIs To Enrich Your Data*.

Summary

In this chapter, you learned the main differences between anonymization and pseudonymization. You also learned which techniques are most frequently used to adopt both of the de-identification processes.

You have also applied the process of anonymization by tokenization and the process of pseudonymization by generating similar pseudonyms in Power BI with both Python and R.

In the next chapter, you will learn how to log data derived from operations done with **Power Query** in Power BI to external repositories.

References

For additional reading, check out the following books and articles:

1. *Pseudonymization, Anonymization and the GDPR* (<https://www.termsfeed.com/blog/gdpr-pseudonymization-anonymization/>)
2. *Symmetric and Asymmetric Encryption* (<https://medium.com/hackernoon/symmetric-and-asymmetric-encryption-5122f9ec65b1>)
3. *Cryptography with Python Tutorial* (https://www.tutorialspoint.com/cryptography_with_python/)
4. *Encryption in R with cyphr* (<https://docs.ropensci.org/cyphr/articles/cyphr.html>)

7 Logging Data from Power BI to External Sources

As you've learned from previous chapters, **Power BI** uses **Power Query** as a tool for **extract transform load (ETL)** operations. The tool in question is really very powerful – it allows you to extract data from a wide variety of data sources and then easily transform it with very user-friendly options in order to persist it into the Power BI data model. It is a tool that is only able to read information from the outside. In fact, the most stringent limitation of Power Query is its inability to write information outside of Power BI. However, thanks to the integration of analytical languages such as **Python** and **R**, you'll be able to persist information about Power Query loading and transformation processes to external files or systems. In this chapter, you will learn the following topics:

- Logging to **CSV** files
- Logging to **Excel** files
- Logging to **Azure SQL Server**

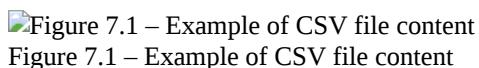
Technical requirements

This chapter requires you to have a working Internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

Logging to CSV files

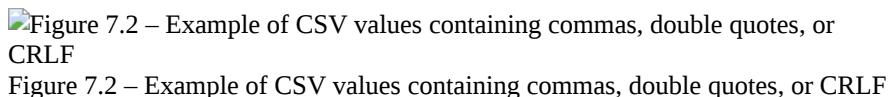
One of the most widely used formats for logging tabular structured information to files is **comma-separated value (CSV)**. Since a CSV file is still a flat text file, CSV is the most popular format for exchanging information between heterogeneous applications.

A CSV file is a representation of a rectangular dataset (**matrix**), containing numeric or string columns. Each row of the matrix is represented by a list of values (one for each column), separated by a comma, and should have the same number of values. Sometimes, other value delimiters could be used, like tab (\t), colon (:), and semi-colon (;) characters. The first row could contain the column header names. Usually, a **line break**, made by **CRLF (Carriage Return Line Feed)** characters (usually entered as \r\n), or simply by **LF** (\n) in Unix systems, is used as a **row delimiter**. So, an example of CSV file content could be the following:



Note the spaces as they become part of a string value! For example, the second value of the v1, v2 row will be [space]v2.

It may happen that a string value contains line breaks (CRLF), double-quotes, or commas (as usually happens with free text fields like “Notes”). In this case, the value should be enclosed in double-quotes and any literal double quote has to be escaped using another double quote. For example, the values " C, D", “E”“F” and "G[CRLF]H" for the column col1 are formatted in the following way in a CSV file:



Important Note

It is important to keep in mind that a CSV file doesn't have any size limits per se.

Since it is a flat text file, the maximum size is determined by the limits imposed by the filesystem. For example, the default filesystem for Windows is the **New Technology File System (NTFS)** and it allows a maximum file

size of *16 TB* as its current implementation. However, its designed theoretical limit is *16 EB* (16×2^{64} bytes) minus *1 KB*.

However, the old **File Allocation Table** filesystem, in its variant of 32 bits (**FAT32**), can only handle a maximum size of *4 GB*.

Important Note

Keep in mind that handling large CSV files leads to memory and/or CPU bottlenecks. You are very likely to get an `OutOfMemory` error if the CSV file to be loaded is larger than the RAM you have available and if your libraries don't use parallelism/distribution/lazy loading mechanisms. You'll learn how to handle such large CSV files in *Chapter 8, Loading Large Datasets Beyond the Available RAM in Power BI*.

Let's see how to read and write CSV files with Python.

Logging to CSV files with Python

Python has a built-in CSV module that provides some functions to read and write CSV files. Very often, however, you will have to import a CSV file whose content will then be transformed through Pandas functions, or you must export in a CSV format a DataFrame previously processed through Pandas. That's why, in these cases, it is much more convenient to directly use the built-in pandas functions with DataFrame objects. You will find the `01-read-csv-file-in-python.py` and `02-write-csv-file-in-python.py` files in the `Chapter07\Python` folder that will show you how to use the CSV module. In the following section, we will focus exclusively on the functionality provided by pandas.

Let's look in detail at the pandas functions available to work with CSV files.

Using the pandas module

The functions that the pandas module provides to read and write a CSV file are very simple and straightforward. You can use the `read_csv()` function to read data from a CSV file. The following is the code that allows you to load the content of the `example.csv` file in a DataFrame:

`Chapter07\Python\03-read-csv-file-with-pandas.py`

```
import pandas as pd
data = pd.read_csv(r'D:\<your-path>\Chapter07\example.csv')
data.head(10)
```

Here is the output using the **VS Code Interactive Window**:

Figure 7.3 – Output of the pandas DataFrame loaded with the example CSV file's content

Figure 7.3 – Output of the pandas DataFrame loaded with the example CSV file's content

The `read_csv` function also allows you to pass the `sep` parameter to define the value separator to be used when reading the file.

If, on the other hand, you need to write a CSV file from the contents of a DataFrame, you can use the `to_csv()` function. The following is an example of the code you could use:

`Chapter07\Python\04-write-csv-file-with-pandas.py`

```
import pandas as pd
data = {
    'Col1' : ['A', 'B', 'C,D', 'E"F', 'G\r\nH'],
```

```

        'Col2' : [23, 27, 18, 19, 21],
        'Col3' : [3.5, 4.8, 2.1, 2.5, 3.1]
    }
data_df = pd.DataFrame(data)
data_df.to_csv(r'D:\<your-path>\Chapter07\example-write.csv', index=False)

```

The `to_csv()` function also allows you to pass the `sep` parameter in order to define the value separator you intend to use in the file.

As you can see, working with CSV files in Python is very easy. In the next section, you'll put this into practice in Power BI.

Logging emails to CSV files in Power BI with Python

As an example of generating a CSV file, we will use the same scenario provided in *Chapter 5, Using Regular Expressions in Power BI*, in which you needed to validate email addresses and ban dates. The goal is to export the rows of the dataset containing an incorrect email to a CSV file and to filter them out of the dataset so that only valid emails remain in Power BI. We will use the `to_csv()` function that pandas provides. The necessary steps are as follows:

1. Follow all the steps in the *Using regex in Power BI to validate emails with Python* section of *Chapter 5, Using Regular Expressions in Power BI* to the end, but do not click on **Close & Apply**.
2. Then, click on **Run Python Script**, enter the following script, and click **OK**:

```

import pandas as pd
filter = (dataset['isEmailValidFromRegex'] == 0)
dataset[filter].to_csv(r'D:\<your-path>\Chapter07\Python\wrong-emails.csv', index=False)
df = dataset[~filter]

```

You can find this Python script also in the `Python\05-log-wrong-emails-csv-in-power-bi.py` file. Note the `~` character that in this case is a negation of the Boolean condition defined by the `filter` variable.

3. Click on the `df` dataset's **Table** value:

Figure 7.4 – Selecting the `df` dataset as a result of the Python script transformation

Figure 7.4 – Selecting the `df` dataset as a result of the Python script transformation

4. Only rows containing valid emails will be kept:

Figure 7.5 – A table containing only valid emails
Figure 7.5 – A table containing only valid emails

Moreover, the `wrong-emails.csv` file has been created in your `Chapter07\Python` folder.

5. Go back to the **Home** menu, and then click **Close & Apply**.

If you go and check the contents of the created CSV file, it matches the following:

```

UserId,Email,BannedDate,IsEmailValidByDefinition,IsDateValidByDefinition,isEmailValidFromRegex
2@example1@example.com@example2@example.com,06/07/2019,0,1,0
3@example33@example.com.,02/05/2018,0,1,0
5@example@example.com --> check,02/29/18,0,0,0
9@example,10/22/2018,0,1,0
13,@example.com,04/24/018,0,0,0
16@example@example.c,6/7/2019,0,1,0

```

As you can see, the emails in the CSV file are all the invalid ones. At this point, you can share the previous file with your colleagues so that they can correct invalid emails.

Well done! You just learned how to log information to a CSV file from Power BI using Python. Now, let's see how you can do the same thing using R.

Logging to CSV files with R

The basic R language provides some out-of-the-box functions for working with CSV files. However, there is also the `readr` package, included in the **Tidyverse** ecosystem, which provides similar functions, but is faster in loading larger CSV files.

Let's see how to use them in detail.

Using the Tidyverse functions

The `readr` package provides some functions mirroring those seen for reading and writing CSV files with R base. The advantage of these functions is that, in addition to respecting the common interface provided by the functions of the Tidyverse world, they are up to five times faster than standard functions and also progress meters. Make sure you have at least version 1.4.0 of the package installed, otherwise, update it.

Always using the usual `example.csv` file, similarly to what we did in the previous section, you can load the data through the `read_csv()` function of the `readr` package in this way:

```
library(readr)
data_df <- read_csv(r'D:\<your-path>\Chapter07\example.csv')
```

As output, you can see the following specification:

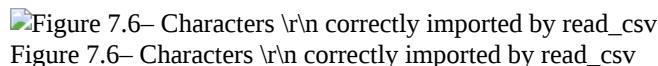
```
-- Column specification -----
cols(
  Col1 = col_character(),
  Col2 = col_double(),
  Col3 = col_double()
)
```

Besides the fact that the `read_csv()` function outputs a **tibble** instead of a `DataFrame`, there is one point that is important to note:

Important Note

The interesting thing is that the `read_csv()` function correctly imports the carriage return character by default.

If you check the newly imported tibble, you have the following:


Figure 7.6– Characters \r\n correctly imported by read_csv

Just as with R base, the `readr` package also provides the same functions, `read_csv2()`, `read_tsv()`, and `read_delim()`, to ensure a similar interface and thus easy usability.

To persist data to a CSV file, the `readr` package provides the `write_csv()` function with its whole family of functions, similarly to R base (`write_csv2`, `write_tsv`, and `write_delim`). Unlike `write.csv()`, these functions do not include row names as a column in the written file. Moreover, the default end-of-line separator is just a new line (`\n`). So, if you want to export your data using the `\r\n` characters as a line separator, you have to pass them through the `eol` parameter:

```
library(readr)
data_df <- data.frame(
  Col1 = c('A', 'B', 'C,D', 'E" F', 'G\r\nH'),
  Col2 = c(23, 27, 18, 19, 21),
  Col3 = c(3.5, 4.8, 2.1, 2.5, 3.1)
```

```
)  
write_csv(data_df, file = r'{D:\<your-path>\Chapter07\R\example-write.csv}', eol = '\r\n')
```

Observe that, in this case, you have to use both characters (`\r\n`) in your data if you want to extract them in exactly the same way.

As you can see, working with CSV files in R is as simple as it was in Python. In the next section, you will log emails and dates with R in Power BI.

Logging dates to CSV files in Power BI with R

We will always use the scenario presented in *Chapter 5, Using Regular Expressions in Power BI*, where it was necessary to validate both email addresses and ban dates. This time, we will use R to export invalid ban dates to a CSV file. The necessary steps are as follows:

1. Follow all the steps in the *Using regex in Power BI to validate dates with R* section of *Chapter 5, Using Regular Expressions in Power BI* to the end, but do not click on **Close & Apply**.
2. Then click on **Run R Script**, enter the following script, and click **OK**:

```
library(readr)  
library(dplyr)  
dataset %>%  
  filter( isDateValidFromRegex == 0 ) %>%  
  write_csv( r'{D:\<your-path>\Chapter07\R\wrong-dates.csv}', eol = '\r\n' )  
df <- dataset %>%  
  filter( isDateValidFromRegex == 1 )
```

You can find this R script also in the `R\01-log-wrong-emails-in-r.R` file.

3. Click on the `df` dataset's **Table** value:

Figure 7.7 – Selecting the `df` dataset as a result of the R script transformation

Figure 7.7 – Selecting the `df` dataset as a result of the R script transformation

4. Only rows containing valid emails will be kept:

Figure 7.8 – A table containing only valid emails
Figure 7.8 – A table containing only valid emails

Moreover, the `wrong-dates.csv` file has been created in your `Chapter07\R` folder.

5. Go back to the **Home** menu, and then click **Close & Apply**.

At this point, you can share the just created `wrong-emails.csv` file with your colleagues so that they can correct invalid emails.

Awesome! You just learned how to log information to a CSV file from Power BI using R. Let's now see how to use Excel files to log your information.

Logging to Excel files

As you probably already know, Microsoft Excel is **spreadsheet** software available in the **Microsoft Office** suite. It's one of the most widely used tools in the world for storing and organizing data in a table format. It is very popular in companies because it allows business data to be shared between departments and enables individual users to do their own data analysis directly and quickly without the help of the IT department.

Early versions of Excel stored information in files of the **Excel Sheet (XLS)** format. This is a proprietary Microsoft format, based on the **Binary Interchange File Format (BIFF)**. It has been the default format for versions from v7.0 (Excel 95) to v11.0 (Excel 2003). From version 8.0 to 11.0 the XLS format can handle $64K$ ($2^{16} = 65,536$) rows and 256 columns (2^8). Starting with version v12.0 (Excel 2007), the default format has changed to **Excel Open XML Spreadsheet (XLSX)**. This is based on the **Office Open XML** format, and it is based on text files that use XML to define all its parameters.

Note

Did you know that an XLSX file contains data in multiple XML files compressed in the **ZIP** format? If you want to verify this, simply rename one of your XLSX files, for example, `example.xlsx`, adding the `.zip` extension to it (for example, `example.xlsx.zip`). Then, extract its content using the **File Explorer** or any other Zip client (like **7-Zip**).

The XLSX format can handle $1024K$ ($2^{20} = 1,048,576$) rows and 16,384 (2^{14}) columns.

Since **Power Pivot** (starting with Excel 2013) and **Power Query** (starting with Excel 2016) were introduced, most of the data ingestion and data analysis activities for the purpose of generating a prototype data model are often performed by power-users thanks to Microsoft Excel. Power Query gives you a set of rich tools for transforming data all in one place. Power Pivot gives you the ability to work with large volumes of data by overcoming Excel's limitation of 1,048,576 rows. Once imported, you can use pivot tables and the **DAX** formula language on them, since the engine behind the scenes is the same as **Analysis Service Tabular** and Power BI. That's why Excel and Power BI are the premier tools for self-service BI on the **Microsoft Data Platform**.

Now, let's see how to interact with Excel files in Python. We will use the latest XLSX format from now on.

Logging to Excel files with Python

The fastest way to interact with Excel files in Python is to use the functions that pandas provides. However, you need to install the `openpyxl` package in your `pbi_powerquery_env` environment. If you remember correctly, you already installed this package in the environment dedicated to **Presidio** (`presidio_env`) in *Chapter 6, Anonymizing and Pseudonymizing your Data in Power BI*. In order to install this package, also in the `pbi_powerquery_env` environment, simply follow these steps:

1. Open your **Anaconda Prompt**
2. Set your current environment to `pbi_powerquery_env`, entering the following command and pressing **Enter**:
`conda activate pbi_powerquery_env`
3. Enter the following command and press **Enter**:
`pip install openpyxl`

As an example, you will find the `example.xlsx` file in the `Chapter07` folder. Let's see how to import its content with Python.

Using the pandas module

You can easily import your data into a pandas DataFrame using this code:

`Chapter07\Python\06-read-excel-file-with-pandas.py`

```
import pandas as pd
data = pd.read_excel(r'D:\<your-path>\Chapter07\example.xlsx')
```

If you visualize the DataFrame in VS Code, you'll see something like the following:



Figure 7.9 – Output of the pandas DataFrame loaded with the example.xlsx file's content

In this case, if an Excel cell contains a string with the `\r\n` characters, the carriage return (`\r`) is lost after import, as you can see in *Figure 7.11*.

As you probably already know, an Excel file (**workbook**) can contain one or more sheets (**worksheets**) in which there is data. If you need to import data from a specific worksheet, you can use this code:

```
data = pd.read_excel(r'D:\<your-path>\Chapter07\example.xlsx', sheet_name='<your-worksheet-name>'
```

Similarly, you can write the contents of a DataFrame to Excel files using this code:

Chapter07\Python\07-write-excel-file-with-pandas.py

```
import pandas as pd
data = {
    'Col1' : ['A', 'B', 'C,D', 'E" F', 'G\r\nH'],
    'Col2' : [23, 27, 18, 19, 21],
    'Col3' : [3.5, 4.8, 2.1, 2.5, 3.1]
}
data_df = pd.DataFrame(data)
data_df.to_excel(r'D:\<your-path>\Chapter07\Python\example-write.xlsx', index = False)
```

The resulting Excel file will have a default `Sheet1` worksheet, and its content will look like the following:



Figure 7.10 – The content of the Excel file created using pandas functions

If you copy the contents of cell `A6` into an advanced editor, you can verify that the `\r\n` characters are kept.

If you want to write the content of your dataset to a specific named worksheet in *a new Excel file*, then you can use the following code:

```
data_df.to_excel(r'D:\<your-path>\Chapter07\Python\example-write-named-sheet.xlsx', sheet_name='My data')
```

The result will be the following:

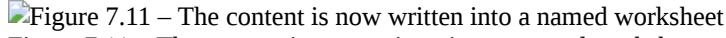


Figure 7.11 – The content is now written into a named worksheet

If instead you want to write the content of your dataset to a specific named worksheet in *an existing Excel file*, then you have to use the pandas `ExcelWriter` class in the following way:

Chapter07\Python\08-write-excel-file-named-sheet-with-pandas.py

```
with pd.ExcelWriter(r'D:\<your-path>\Chapter07\Python\example-write-named-sheet.xlsx', mode='a')
    data_df.to_excel(writer, sheet_name='My data', index = False)
```

Note, that the `mode='a'` is for “*append*”. Let's now look at an example of logging in Power BI using the previous pandas functions.

Logging emails and dates to Excel files in Power BI with Python

Let's go back to the same scenario used in the previous sections, namely, the one we already analyzed in *Chapter 5, Using Regular Expressions in Power BI*, in which you needed to validate email addresses and ban dates. This time, however, the goal is to export invalid emails and invalid dates to two separate worksheets in an Excel file and then share it with the team.

Now, open your Power BI Desktop, make sure the Python environment to use is `pbi_powerquery_env`, and let's get started:

1. From the ribbon, click on the **Excel** icon to import data from Excel.
2. From the **Open** dialog box, select the `users.xlsx` file you can find in the `Chapter05` folder.
3. From the **Navigator** window, select the **Users** sheet and then click on **Transform Data**.
4. Click on the **Transform** menu, and then click on **Run Python Script**.
5. Copy the code you can find in the `09-validate-emails-dates-with-regex-in-power-bi.py` file in the `Chapter07/Python` folder, and paste it into the Python script editor. This code is just a merging of the scripts you already used to validate emails and dates separately. Then, click **OK**.

6. Select just the **Table** value related to the `df` table name and you will see something like this:

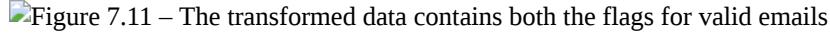


Figure 7.11 – The transformed data contains both the flags for valid emails

and dates

Figure 7.11 – The transformed data contains both the flags for valid emails and dates

Now, you have both the `isEmailValidFromRegex` and `isValidDateFromRegex` flags that allow you to select emails and correct dates.

7. Click again on **Run Python Script**, enter the script you can find in the `10-log-wrong-emails-dates-excel-in-power-bi.py` file, and click **OK**.
8. Select just the **Table** value related to the `df` table name and you will see a table where only rows containing valid emails and dates will be kept:



Figure 7.12 – The output data contains rows containing valid emails and

dates

Figure 7.12 – The output data contains rows containing valid emails and dates

Moreover, the `wrong-data.xlsx` file has been created in your `Chapter07/Python` folder and it contains two worksheets: `Wrong emails` and `Wrong dates`.

9. Go back to the **Home** menu, and then click **Close & Apply**.

Amazing! You just learned how to log information to an Excel file in multiple sheets from Power BI using Python. Now, let's see how you can do the same thing using R.

Logging to Excel files with R

To be able to read and write Excel files in R, we recommend the use of two separate packages:

- **readxl**: This is a package that is part of the Tidyverse world and allows you to read the information contained in an Excel file in the simplest and most flexible way.
- **openxlsx**: This is a package that provides a high-level interface for creating and editing Excel files. Compared to other packages that do the same thing, `openxlsx` removes the dependency on **Java** behind the scenes.

First, you need to install the `openxlsx` package:

1. Open **RStudio**, and make sure your latest **MRO** engine is selected in the **Global Options** (in our case, MRO 4.0.2).
2. Enter the `install.packages("openxlsx")` command in the console. Remember that it installs the package at the version corresponding to the **CRAN** snapshot defined by your MRO installation.

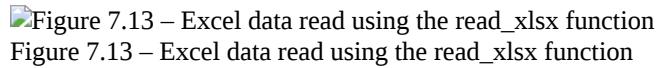
Now you are ready to learn how to read and write data to Excel files.

Using the `readxl` and `openxlsx` packages

The `readxl` package provides two separate functions – `read_xls()`, to read Excel files in an XLS format, and `read_xlsx()`, to read those in an XLSX format. If you want to read the contents of the `example.xlsx` file located in the `Chapter07` folder, you can use the following code:

```
library(readxl)
data_tbl <- read_xlsx(r'{D:\<your-path>\Chapter07\example.xlsx'})
```

The result will be a tibble:



As you can see, the carriage returns and the line feed characters (`\r\n`) are kept. If you want to read data from a specific worksheet instead, you can use this code:

```
data_tbl <- read_xlsx(r'{D:\<your-path>\Chapter07\example.xlsx}', sheet = 'My sheet')
```

To write your data into Excel files, you can use the `write.xlsx()` function of the `openxlsx` package, as follows:

```
library(dplyr)
library(openxlsx)
data_df <- data.frame(
  Col1 = c('A', 'B', 'C,D', 'E,F', 'G\nH'),
  Col2 = c(23, 27, 18, 19, 21),
  Col3 = c(3.5, 4.8, 2.1, 2.5, 3.1)
)
data_df %>%
  write.xlsx(file = r'{D:\<your-path>\Chapter07\R\example-write.xlsx}', colNames = TRUE)
```

Observe how, in this case, you have to use the “Unix convention” regarding the new lines, and that is to use only the `\n` character in the strings of your data to have the standard Windows characters `\r\n` in Excel.

If you want to write the content of your dataset to a specific named worksheet in *a new Excel file*, then you can use the following code:

```
data_df %>%
  write.xlsx(file = r'{D:\<your-path>\Chapter07\R\example-write.xlsx}', colNames = TRUE, sheetName = 'My sheet')
```

If, on the other hand, you need to add a worksheet to *an existing Excel file*, you have to use a named list of DataFrames/tibbles as the input of the `write.xlsx` function. This is the code to use if you can manually assign a string name to each sheet:

```
df_named_lst <- list("My data 1" = data_df, "My data 2" = data_df)
write.xlsx(df_named_lst, file = r'{D:\<your-path>\Chapter07\R\example-write.xlsx'})
```

Keep in mind that if you need to use a list of DataFrames/tibbles (`df_lst`) and a list of worksheet names (`names_lst`) separately, you can use the following code to write all your data in one Excel workbook:

```
df_named_lst <- setNames(df_lst, names_lst)
write.xlsx(df_named_lst, file = r'{D:\<your-path>\<your-file>.xlsx'})
```

Let's now look at an example of logging in Power BI using the previous R functions.

Logging emails and dates to Excel in Power BI with R

The example we will use is still the one from *Chapter 5, Using Regular Expressions in Power BI*, in which you needed to validate email addresses and ban dates. The ultimate goal will always be to export invalid emails and invalid dates to two separate worksheets in an Excel file and then share it with the team.

Now open your Power BI Desktop, make sure your latest MRO is referenced in the options, and let's get started:

1. From the ribbon, click on the **Excel** icon to import data from Excel.
2. From the **Open** dialog box, select the `users.xlsx` file you can find in the `Chapter05` folder.
3. From the **Navigator** window, select the **Users** sheet and then click on **Transform Data**.
4. Click on the **Transform** menu, and then click on **Run R Script**.
5. Copy the code you can find in the `02-validate-emails-dates-with-regex-in-power-bi.R` file in the `Chapter07/R` folder, and paste it into the R script editor. This code is just a merging of the scripts you already used to validate emails and dates separately. Then click **OK**.
6. Select just the **Table** value related to the `df` table name and you will see something like this:

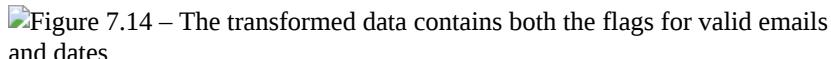


Figure 7.14 – The transformed data contains both the flags for valid emails and dates

Now you have both the `isEmailValidFromRegex` and `isValidDateFromRegex` flags that allow you to select correct emails and dates.

7. Click again on **Run R Script**, enter the script you can find in the `03-log-wrong-emails-dates-excel-in-power-bi.R` file, and click **OK**. Remember to change the paths in the code.
8. Select just the **Table** value related to the `df` table name and you will see a table where only rows containing valid emails and dates will be kept:



Figure 7.15 – The output data contains rows containing valid emails and dates

Moreover, the `wrong-data.xlsx` file has been created in your `Chapter07/R` folder and it contains two worksheets: `Wrong emails` and `Wrong dates`.

9. Go back to the **Home** menu, and then click **Close & Apply**.

Awesome! You just learned how to log information to an Excel file in multiple sheets from Power BI using R.

In the next section, you will learn how to log information from Power BI to either an **on-premises SQL Server**, or to an **Azure SQL Server**.

Logging to an Azure SQL Server

In the vast majority of companies, business information is persisted in a **Relational Database Management System (RDBMS)**. Microsoft's quintessential relational database is **SQL Server** in its on-premises version if the company adopts the Microsoft Data Platform. Otherwise it is **Azure SQL Server** which is a **Platform as a Service (PaaS)**, cloud-hosted database.

Generally, it is a good idea to centralize all of a company's key information in a single repository. That's why it might be useful to know how to log information from within a Power BI process into a SQL Server database or an Azure SQL database.

If you have the option to already access an instance of SQL Server on-premises or an Azure SQL Server, you just need to make sure that the **ODBC Driver for SQL Server** is installed on your machine. In fact, both Python and R will connect to (Azure) SQL Server via an ODBC connection. You have the option to install the driver on your machine directly (via the link <http://bit.ly/ODBC-SQLServer>), but more often this driver is installed indirectly when installing the ultimate client for managing any SQL infrastructure, which is **SQL Server Management Studio (SSMS)**.

On the other hand, if you don't have access to either an on-premises SQL Server instance or an Azure SQL database, then you have two options for testing the examples in this section:

- Install a free instance of **SQL Server Express Edition** (or **Developer**).
- Create an Azure SQL database from the Azure portal using your account.

Let's see in detail how to proceed for each of these options.

Installing SQL Server Express

In this section, we will show how to install the Express Edition of SQL Server. This is the free version of Microsoft's database engine that can also be used in production for desktop and small server data-driven applications. Obviously, the Express Edition has limitations that distinguish it from the top-of-the-line **Enterprise Edition**. Here are a few examples:

- A maximum memory of 1,410 MB is used by an instance of the database engine.
- A maximum size of 10 GB for a single database.
- Compute capacity used by a single instance limited to the lesser of one socket or four cores.

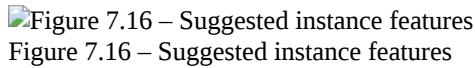
Despite these limitations, SQL Server Express remains an excellent solution to use in production for small applications. If, on the other hand, you need to be able to test the more advanced features of the engine because you know that your application will use a more complete edition in production (Standard or Enterprise), you can install the **Developer Edition** on your machine. This edition allows you to test all the features of the Enterprise Edition while not paying for an Enterprise license. The most stringent limitation is, of course, that the Developer Edition cannot be used in production.

That said, there are tons of tutorials about how to install the latest version of SQL Server Express available to date (2019). One of the many that we suggest you follow for installation is this one:

<https://www.sqlshack.com/how-to-install-sql-server-express-edition>

Just keep these observations in mind:

1. The **Feature Selection** screen suggests also installing the **SQL Server Replication**. If you would also like to test Machine Learning Services separately with R, Python and Full-Text available in SQL Server, we suggest selecting the following, leaving out the Replication and Java options:



The important thing is to keep the **Shared Features** selected by default so that the ODBC drivers needed to connect to the instance are also installed.

2. Remember to save the password for the sa (system administrator) user in a safe place, because it provides access to the instance you are installing as an administrator.
3. The tutorial will ask you to install SSMS in *Step 5*. If you haven't already installed it, do it now.
4. In order to connect to your SQL Server Express instance with SSMS, instead of the computer name and then the instance name, you can also use .\SQLEXPRESS as the **Server name**. That said, the tutorial suggests testing your connection using the **SQL Server Authentication** with the **sa** account credentials. Remember that you can also connect to your instance directly using the **Windows authentication**, as your user is automatically added to the **SQL Server Administrators** group. As the tutorial says, if the login window closes without any issues after clicking **Connect**, this means the connection works properly.

Just stop at the *For the Windows authentication* section. The following applies:

Important Note

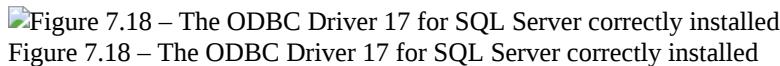
It is important that the connection to the new SQLExpress instance works properly from SSMS. This confirms the correct installation of the instance.

After the installation is complete, you can verify that the ODBC drivers have been installed by following these steps:

1. Click the Windows **Start** button, and start entering the string **ODBC**. You will see something like this:



2. Click on the 32-bit app or the 64-bit app, and then click on the **Drivers** tab. You'll see the **ODBC Driver 17 for SQL Server** installed:



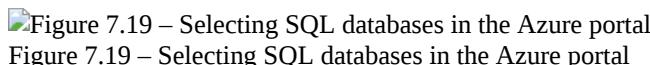
Fantastic! Now your SQLExpress instance is working properly.

Let's also see how to configure an Azure SQL database through the Azure portal.

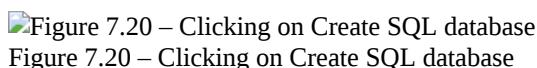
Creating an Azure SQL database

In order to create an Azure SQL database, you must be subscribed to Azure services. This gives you the ability to access the **Azure portal** (<https://portal.azure.com/>) to manage all of your tenant's Azure resources. Once you have accessed the portal, follow these steps:

1. Search for **SQL databases** in the main search box and click on it:

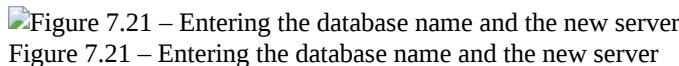


2. Click on **Create SQL database** in the middle of the page:

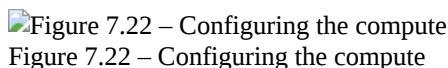


3. You need to select a **Subscription** associated with your account (usually, the selected default one is ok if you have just one subscription associated) and then a **Resource group** to collect all the resources you want to create under a name (like a virtual folder). Select one if you have already created it, or create a new one if needed.

4. You must also provide the **Database name** and **Server**. As the database name, use **SystemLogging**. If this is your first time creating an Azure SQL database, you need to create a new server too. So, click on **Create new** and provide a server name, a login, the related password, and the location you prefer. After clicking **OK**, you'll see something like this:



5. As we're creating an Azure SQL database for testing purposes, we can choose a **Standard Database Transaction Unit (DTU)** workload. So, click on **Configure database** under **Compute + storage**:



The **General Purpose** option is selected by default. So, click on the **Looking for basic, standard, premium?** link on the top-left and then click on the **Standard** workload option. Then, click on the **Apply**

button.

6. Select also to install sample data (from the demo **AdventureWorkLT** database) in the **Additional settings** tab by clicking on **Sample**:

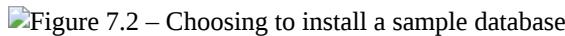


Figure 7.2 – Choosing to install a sample database

7. Click on **Review + create**, and then on **Create** to deploy your brand-new Azure SQL database.

8. After the deployment is complete, click on **Go to resource** to access the general dashboard of the newly created resource. In the top-right corner, you will notice that the server's name takes the following form:

<your-server-name>.database.windows.net

9. In order to access your Azure SQL database from any client, you must declare the client's IP address in the server firewall rules. To do this, click on **Set server firewall** at the top of the dashboard:

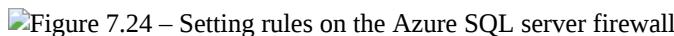


Figure 7.24 – Setting rules on the Azure SQL server firewall

In addition to other options, you'll see textboxes to enter the **Rule name**, the **Start IP**, and the **End IP**.

Additionally, the portal also shows you the IP address from which you are currently connected:



Figure 7.25 – Copying your current IP address and using it in your rule

You can then enter your Client IP address as the Start and End IP if you will be connecting to the Azure SQL database from that same machine. Keep in mind that if your machine's public IP address is not static, it will assume a new IP address on its next reboot that is different from the previous one. Therefore, if you need to connect to your Azure SQL database from your machine often, make sure to create a **static public IP address**. If you have just created a new Azure Virtual Machine, Azure will ask you if you want to make its current public IP address static when you stop it for the first time. If you decide yes, Azure will do everything automatically. Otherwise, you can easily configure it later by following this tutorial:<http://bit.ly/AzureVM-assign-static-ip> Click **Save** to keep your changes in the firewall settings.

10. At this point, you can test the connection to your new Azure SQL database using SSMS. If you haven't installed it yet, do so now by downloading the installer from the link <http://aka.ms/ssms>. Once installed, open the **Microsoft SQL Server Management Studio** app from the **Start** menu (you can find it under the **Microsoft SQL Server Tools XX** folder). Use the server name in the <your-server-name>.database.windows.net format, choose **SQL Server Authentication** as the authentication method, and then enter the login and password you used during the creation of your Azure SQL database. Then click **Connect**:

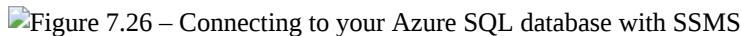


Figure 7.26 – Connecting to your Azure SQL database with SSMS

11. If the **Connect to Server** window disappears, then you are connected to your server. In fact, if you open the **Databases** node in the **Object Explorer** on the left, you can see your **SystemLog** database containing the **AdventureWorkLT** tables:



Figure 7.27 – You are connected to your Azure SQL database

Awesome! Your Azure SQL database is running and ready to be used.

Let's try now to read and write data into a SQL Server or an Azure SQL database using Python.

Logging to an Azure SQL server with Python

The most widely used Python module for connecting to databases to which you can connect via ODBC drivers is **pyodbc**. So, first you need to install this package in the `pbi_powerquery_env` environment:

1. Open your Anaconda Prompt.
2. Switch to the `pbi_powerquery_env` environment by entering this command:

```
conda activate pbi_powerquery_env
```

3. Install the new package by entering this command:

```
pip install pyodbc
```

At this point, you can start interacting with your database instances using Python.

Using the pyodbc module

First, you need to *create a connection* to your database instance. You can do this with the `connect()` function, which accepts a *connection string* as an argument. Depending on whether you need to connect to an instance of a SQL server on-premises or an Azure SQL database, the connection string varies only in its `server` parameter.

You can establish a connection to your on-premises instance using the Windows authentication with this code:

```
import pyodbc
conn = pyodbc.connect(
    'Driver={ODBC Driver 17 for SQL Server};'
    'Server=.\SQLEXPRESS;'
    'Database=master;'
    'Trusted_Connection=yes;')
```

You can also find the code snippets used here in the `11-read-write-on-azure-sql-server-with-python.py` file in the `Chapter07\Python` folder. In contrast to the other Python script you found in the repository, in this one you can find comments like `# %%`. They are placeholders that VS Code recognizes as **Jupyter-like code cells**. When VS Code identifies a cell, it automatically adds commands to execute its contents in the Interactive Window, making it easier for the user to interact with the code.

If you want to connect to the same instance using the SQL authentication instead, you can use this code:

```
import pyodbc
conn = pyodbc.connect(
    'Driver={ODBC Driver 17 for SQL Server};'
    'Server=.\SQLEXPRESS;'
    'Database=master;'
    'Uid=sa;'
    'Pwd=<your-password>')
```

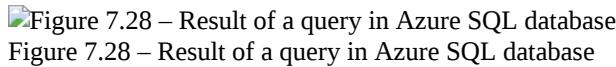
The format of the previous connection strings remains the same even when you want to connect to an Azure SQL database. You must use the format `<your-server-name>.database.windows.net` as the server name. The authentication mode must necessarily be the SQL authentication mode. Therefore, the code to connect to your Azure SQL database is as follows:

```
import pyodbc
conn = pyodbc.connect(
    'Driver={ODBC Driver 17 for SQL Server};'
    'Server=<your-server-name>.database.windows.net;'
    'Database=SystemsLogging;'
    'Uid=<your-username>;'
    'Pwd=<your-password>')
```

Once you have established a connection to an instance of your choice, you can read data from tables or views via the pandas `read_sql()` function, which accepts a query in SQL (in our case, in **T-SQL** for SQL Server) as a parameter. For example, regardless of whether you are connected to your on-premises instance or on Azure, you can run the following code to read the database information available in the instance:

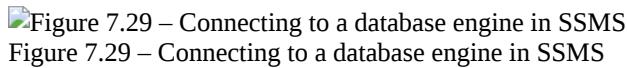
```
import pandas as pd
data = pd.read_sql("SELECT database_id, name FROM sys.databases", conn)
data.head()
```

In the case of Azure SQL, you will see this result:

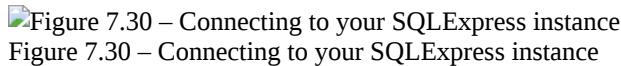


Let's try writing something to a database instead. First, in the case of your on-premises instance, you need to create a new database to write your data to. You can do that in SSMS following these steps:

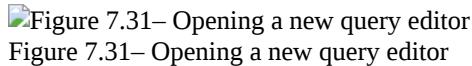
1. Click on **Connect**, and then on **Database Engine....**:



2. Connect to your SQLExpress instance using the Windows authentication with the string `.\\SQLExpress` as the server name. Then, click on **Connect**:

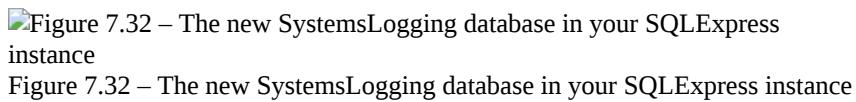


3. Click on **New Query** on the toolbar at the top:



4. Enter the `CREATE DATABASE SystemsLogging` script and then click on the **Execute** button with the green arrow (or just press **F5**).

5. Open the **Databases** node in the **Object Explorer** and you can now see the brand-new `SystemsLogging` database:



Now you can create the new `WrongEmails` table in the `SystemsLogging` database. It is usually preferable to run **Data Definition Language (DDL)** commands (like `CREATE`) directly in SSMS. In this case, we'll do it through Python to show you some special commands. You will first create a `cursor` object from the `conn` connection and then call its `execute()` method, passing it a `CREATE TABLE` query:

```
conn = pyodbc.connect(
    'Driver={ODBC Driver 17 for SQL Server};'
    r'Server=.\SQLExpress;'
    'Database=SystemsLogging;'
    'Trusted_Connection=yes;')
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE WrongEmails
    (
        UserId int,
        Email nvarchar(200)
    )
''')
conn.commit()
```

Keep in mind that an Azure SQL Server database collects its objects (tables, views, and so on) in **SQL schemas**. Usually, when you create a database object, you also specify the schema in the CREATE statement. For a table, you generally use the CREATE TABLE <your-schema>. <table-name> script. The WrongEmails table was created without specifying any schema. Therefore, it assumes the default schema, which is dbo.

Make sure to create the same table in your Azure SQL database:

```
conn = pyodbc.connect(  
    'Driver={ODBC Driver 17 for SQL Server};'  
    'Server=<your-server-name>.database.windows.net;'  
    'Database=SystemsLogging;'  
    'Uid=<your-username>;'  
    'Pwd=<your-password>')  
cursor = conn.cursor()  
cursor.execute(''  
    CREATE TABLE WrongEmails  
    (  
        UserId int,  
        Email nvarchar(200)  
    )  
'')  
conn.commit()
```

At this point, you can *write pandas DataFrame content* in the WrongEmails table row by row, using the cursor.execute() method, passing to it an INSERT INTO query. We will use the Azure SQL database in the example since there is also the SalesLT.Customer table there (note the SalesLT schema) from which to read some customer data to write to the WrongEmails table:

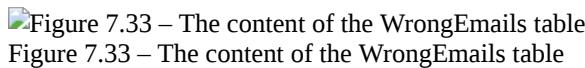
```
# Get data from sample Customers  
data = pd.read_sql('SELECT TOP 10 CustomerID, EmailAddress FROM SalesLT.Customer', conn)  
# Write Customers data into the WrongEmails table  
cursor = conn.cursor()  
# Write a dataframe into a SQL Server table row by row:  
for index, row in data.iterrows():  
    cursor.execute("INSERT INTO WrongEmails (UserId, Email) values(?,?)", row.CustomerID, row.Email)  
conn.commit()  
cursor.close()
```

The iterrows() function iterates over the DataFrame columns, and it will return a tuple with the column name and content in the form of series. Keep in mind that if you want to write to a SQL server on-premises, you only need to change the connection string, and the syntax you just saw remains valid. You can't run the code exactly as it is written in your SQLExpress instance simply because there is no AdventureWorksLT example data there, so it will give you an error.

To display the first rows of the WrongEmails table, you can use this code:

```
df = pd.read_sql('SELECT TOP 10 UserId, Email FROM WrongEmails', conn)  
df.head()
```

You will see something like this in VS Code:



Now, make sure to empty the WrongEmails table with the following command so that it can be ready to be used later:

```
cursor = conn.cursor()  
cursor.execute('TRUNCATE TABLE WrongEmails')  
conn.commit()
```

When you have finished all read and write operations on the database instance, remember to close the connection as follows:

```
conn.close()
```

Hey! You just learned how to read and write data from a SQL Server or Azure SQL database via Python. Simple, isn't it? Let's apply what you've learned in Power BI.

Logging emails and dates to an Azure SQL database in Power BI with Python

In this section, we will use the same scenario provided in *Chapter 5, Using Regular Expressions in Power BI*, in which you validated email addresses and ban dates. The goal is to log the rows of the dataset containing incorrect emails to an Azure SQL database and to filter them out of the dataset so that only valid emails remain in Power BI. In order to properly execute the following Python code, you need to make sure you have created both the Azure SQL SystemsLogging database and the WrongEmails table as discussed in the previous section. If you prefer, you can also use your on-premises SQL Server instance by appropriately changing the server name in the connection string. In this case, make sure that the SystemsLogging database and the WrongEmails table are there.

The necessary steps are as follows:

1. Follow all the steps in the *Using regex in Power BI to validate emails with Python* section of *Chapter 5, Using Regular Expressions in Power BI*, to the end, but do not click on **Close & Apply**.
2. Then click on **Run Python Script**, enter the script you can find in the `Python\12-log-wrong-emails-azure-sql-in-power-bi.py` file, and click **OK**.
3. Click on the `df` dataset's **Table** value.
4. Only rows containing valid emails will be kept:

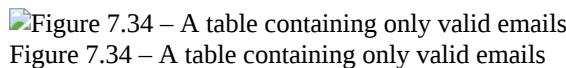


Figure 7.34 – A table containing only valid emails

Moreover, the invalid emails have been written into the `WrongEmails` table of your `SystemsLogging` Azure SQL database.

5. Go back to the **Home** menu, and then click **Close & Apply**.

To verify that invalid emails were indeed written to the previous table, you can do this with SSMS:

1. Connect with SSMS to your Azure SQL database using the `<your-server-name>.database.windows.net` string as **Server name** and the SQL authentication.
2. Open the **Databases** node, then open the **Tables** node, right-click on the **dbo.WrongEmails** table, and click on **Select Top 1000 Rows**:

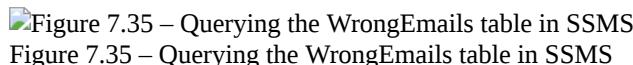


Figure 7.35 – Querying the WrongEmails table in SSMS

You'll see the following output:

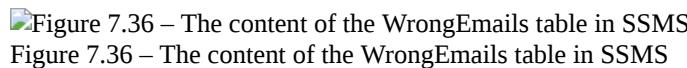


Figure 7.36 – The content of the WrongEmails table in SSMS

Now, third-party systems can simply access your Azure SQL database (even simply with Excel, see the *References* section) to read invalid emails and mobilize the appropriate team to correct them.

Well done! You just learned how to log information to an Azure SQL database from Power BI using Python (you can do the same writing into an on-premises SQL Server database just by changing the connection string). Now, let's see how you can do the same using R.

Logging to an Azure SQL server with R

To connect to a database via ODBC drivers in R, we will use two packages: **DBI** and **odbc**. The **DBI** package has the task of separating the connectivity to the database into a *frontend* and a *backend*. The R code you'll write will use only the exposed frontend API. The backend will take care of communicating with the specific DBMS through special drivers provided by the installation of other packages. In our case, the **odbc** package will allow us to interface with SQL Server instances, both on-premises and on Azure. So, first you need to install these packages in your most recent MRO engine:

1. Open RStudio, and make sure it is referencing your latest MRO in the **Global Options**.

2. Enter this command into the console:

```
install.packages(c('odbc', 'DBI'))
```

At this point, you can start interacting with your database instances using R.

Using the **DBI** and **odbc** packages

Also, in this case, you need to *create a connection* to your database instance. You can do this with the **dbConnect()** function of the **DBI** package, which accepts a *driver object* (in our case the **odbc()** one from the **odbc** package) and a *connection string* as arguments.

You can establish a connection to your **SQLExpress** on-premises instance using the Windows authentication with this code:

```
R\04-read-write-on-azure-sql-server-with-r.R
```

```
library(odbc)
library(DBI)
conn <- dbConnect(
  odbc::odbc(),
  server = r'`.\\SQLEXPRESS`',
  database = 'SystemsLogging',
  trusted_connection = 'yes',
  driver = '{ODBC Driver 17 for SQL Server}'
)
```

If you want to connect to the same instance using the SQL authentication instead, you can use this code:

```
conn <- dbConnect(
  odbc::odbc(),
  server = r'`.\\SQLEXPRESS`',
  database = 'SystemsLogging',
  uid = 'sa',
  pwd = '<your-password>',
  driver = '{ODBC Driver 17 for SQL Server}'
)
```

The format of the previous connection strings remains the same even when you want to connect to an Azure SQL database. You just need to use the `<your-server-name>.database.windows.net` format as the server name. In this case, the authentication mode must necessarily be the SQL authentication one. Therefore, the code to connect to your Azure SQL database is as follows:

```
conn <- dbConnect(
  odbc::odbc(),
  server = '<your-server>.database.windows.net',
  database = 'SystemsLogging',
  uid = '<your-username>',
  pwd = '<your-password>',
  driver = '{ODBC Driver 17 for SQL Server}'
)
```

An interesting thing is that, as soon as the connection is established, RStudio allows you to browse the databases inside your server through the **Connections** tab at the top right. For example, *Figure 7.37* displays the contents of the **SystemLogging** database of the **SQLExpress** instance, going down to the detail of individual columns in a table:



Figure 7.37 – RStudio's object explorer of the connection established

Figure 7.37 – RStudio's object explorer of the connection established

Remember that the `WrongEmails` table was created in the previous section.

Once you have established a connection to an instance of your choice, you can easily read data from tables or views via the DBI's `dbGetQuery()` function, which accepts a SQL query (in our case, T-SQL for SQL Server) as a parameter. For example, in the same way as with Python, you can run the following code to read the database information available on both on-premises instances and Azure SQL servers:

```
data <- DBI::dbGetQuery(conn, "SELECT database_id, name FROM sys.databases")
head(data)
```

In the case of Azure SQL, you will see this result:

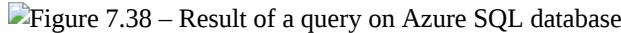


Figure 7.38 – Result of a query on Azure SQL database

Let's try writing something to a database instead. For example, you can *write a R DataFrame content* into the `WrongEmails` table using the `dbAppendTable()` method. It simply accepts the connection object, the name of the target table, and the DataFrame containing the source data. You just need to be careful to properly rename the source data columns using aliases in the SQL query (using the `AS` keyword) when reading from the database, so that the names match those of the target table columns. We will use an Azure SQL database in the example since there is also the `SalesLT.Customer` table from which to take the source data:

```
customers_df <- dbGetQuery(conn, "SELECT TOP 10 CustomerID AS UserId, EmailAddress AS Email FROM
dbAppendTable(conn, name = 'WrongEmails', value = customers_df)
```

Keep in mind that if you need to create the target table and fill it, you can use the one-shot `dbCreateTable()` method that get the same parameters of the `dbAppendTable()` method. To display the first rows of the `WrongEmails` table, you can use this code:

```
df <- dbGetQuery(conn, "SELECT TOP 10 UserId, Email FROM WrongEmails")
head(df)
```

You will see something like this in VS Code:



Figure 7.39 – The content of the WrongEmails table

Now make sure to empty the `WrongEmails` table with the `TRUNCATE TABLE` SQL command inside the `dbSendQuery()` method (which just executes a query without retrieving any data), so that it is ready to be used:

```
dbSendQuery(conn, "TRUNCATE TABLE WrongEmails")
```

When you have finished all read and write operations on the database instance, remember to close the connection, as follows:

```
dbDisconnect(conn)
```

Wow! You just learned how to read and write data from a SQL server or Azure SQL database with R! Let's apply what you've learned in Power BI.

Logging emails and dates to an Azure SQL database in Power BI with R

In this section, we will use the same scenario already used to show how to log data to Azure SQL from Power BI. In order to properly execute the following R code, you need to make sure you have created the Azure SQL `SystemsLogging` database and the `WrongEmails` table as discussed in the *Logging to an Azure SQL server with Python* section. If you prefer, you can also use your on-premises SQL Server instance by appropriately changing the server name in the connection string. In this case, make sure that the `SystemsLogging` database and the `WrongEmails` table are there. The necessary steps are as follows:

1. Follow all the steps in the *Using regex in Power BI to validate emails with R* section of *Chapter 5, Using Regular Expressions in Power BI*, to the end, but do not click on **Close & Apply**.
2. Then, click on **Run R Script**, enter the script you can find in the `R\05-log-wrong-emails-azure-sql-in-power-bi.R` file, and click **OK**. Remember to edit the server's name and your credentials in the code.
3. Click on the `df` dataset's **Table** value.

4. Only rows containing valid emails will be kept:



Figure 7.40 – A table containing only valid emails

Moreover, the invalid emails have been written into the `wrongEmails` table in your Azure SQL database.

5. Go back to the **Home** menu, and then click **Close & Apply**.

As done before, you can verify that invalid emails were written to the previous table with SSMS. You'll see the following output:

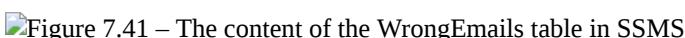


Figure 7.41 – The content of the `WrongEmails` table in SSMS

Awesome! You've just logged your wrong emails to your (Azure) SQL Server database using R from Power BI.

Summary

In this chapter, you learned how to log some information processed in Power Query to CSV files, Excel, on-premises SQL Servers, and Azure SQL, in both Python and R, using very simple and straightforward commands.

In the next chapter, you will see how to handle very large CSV files that cannot be loaded from Power BI Desktop due to the RAM size of your machine not being sufficient.

References

For additional reading, check out the following books and articles:

1. *Common Format and MIME Type for Comma-Separated Values (CSV) Files* (<https://tools.ietf.org/html/rfc4180>)
2. *Excel (.xlsx) Extensions to the Office Open XML SpreadsheetML File Format* (https://docs.microsoft.com/en-us/openspecs/office_standards/ms-xlsx/)
3. *Connect Excel to a database in Azure SQL Database* (<https://docs.microsoft.com/en-us/azure/azure-sql/database/connect-excel>)

8 Loading Large Datasets beyond the Available RAM in Power BI

In the previous chapter, you learned how to read from and write to a CSV file, both with Python and in R. When it comes to reading a file, whether you use Power BI's standard data import feature or the techniques shown in the previous chapter, the main limitation on the file size is due to the amount of RAM available on the machine where Power BI Desktop is installed.

In a data enrichment phase, it may be necessary to extract information needed for ongoing analysis from very large files (terabytes in size). In these cases, it is almost always necessary to implement big data solutions to be able to handle such masses of data. Very often, however, it is necessary to import files that are slightly larger than the available RAM, in order to extract aggregate information and then persist it in a small table for reuse during processing. In such cases, it's not necessary to bother with demanding big data platforms, but you can take advantage of the flexibility provided by specific packages that implement distributed computing systems in both Python and R, without having to resort to Apache Spark-based backends.

In this chapter, you will learn the following topics:

- A typical analytic scenario using large datasets
- Importing large datasets with Python
- Importing large datasets with R

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

A typical analytic scenario using large datasets

One of the most frequent activities of a data scientist is to analyze a dataset of information relevant to a business scenario. The objective of the analysis is to be able to identify associations and relationships between variables, which help in some way to discover new measurable aspects of the business (insights) and can then be used to make it grow better. It may be the case that the available data may not be sufficient to determine strong associations between variables, because any additional variables may not be considered. In this case, attempting to obtain new data that is not generated by your business but enriches the context of your dataset (a **data augmentation** process) can improve the strength of the statistical associations between your variables. Being able to link, for example, weather forecast data to a dataset that reports the measurements of the water level of a dam certainly introduces significant variables to better interpret the phenomenon.

It is in this context that you often find yourself having to extract information from CSV files downloaded from external data sources. For example, imagine that you have been assigned the task of analyzing what factors influenced the profitability of a chain of shoe stores located in major airports in the United States from 1987 to 2012. The first thing that comes to your mind is that maybe flight delays are somehow related to people staying at the airport. If you have to spend time at the airport, you definitely have more time to visit the various stores there and therefore the chance of you making a purchase increases. So, how do you find data on the average airline delay at each US airport for each day of the year? Fortunately, the *Research and Innovative Technology Administration (RITA), Bureau of Transportation Statistics*, provides aggregated statistics (<http://bit.ly/airline-stats>) and raw data containing flight arrival and departure details for all commercial flights within the US (<http://bit.ly/airline-stats-data>). A set of CSV files containing monthly airline data from 1987 to 2012 is already collected and zipped by Microsoft, and you can download it directly from this link: <http://bit.ly/AirOnTime87to12>.

If you would like more information about the fields in the files, please see the `AirOnTime87to12.dataset.description.txt` file.

The compressed file in question is about 4 GB large and, once unzipped, contains many CSV files with detailed data of flights made in the US across months ranging from 1987 to 2012, with a total size of 30 GB! Your goal is to calculate the average daily flight delay for each origin airport and to save the resulting dataset in a CSV file. How do you import all that data in Power BI!? Let's see how to do this in Python.

Import large datasets with Python

In *Chapter 3, Configuring Python with Power BI*, we suggested that you install some of the most commonly used data management packages in your environment, including NumPy, pandas, and scikit-learn. The biggest limitation of these packages is that *they cannot handle datasets larger than the RAM of the machine in which they are used*, thus they are not able to scale to more than one machine. To comply with this limitation, distributed systems based on **Spark**, which has become a dominant tool in the big data analysis landscape, are often used. However, the move to these systems forces developers to have to rethink already written code using an API called **PySpark**, born to use Spark objects with Python. This process is generally seen as causing delays in project delivery and causing frustration for developers, who master the libraries available for standard Python with much more confidence.

In response to the preceding issues, the community developed a new library for parallel computing in Python called **Dask** (<https://dask.org/>). This library provides transparent ways for the developer to scale pandas, scikit-learn, and NumPy workflows more natively, with minimal rewriting. Dask APIs are pretty much a copy of most of the APIs of those modules, making the developer's job easier.

One of the advantages of Dask is that *you don't need to set up a cluster of machines to be able to manipulate 100+ GB datasets*. You just need one of today's laptops with a multi-core CPU and 32 GB of RAM to handle them with ease. Therefore, thanks to Dask, you can perform analysis of moderately large datasets on your own laptop, without incurring the overhead typical of clusters, such as the use of Docker images on various nodes and complex debugging.

Important Note

Evidently, even the Spark team realized the *inconvenient* points born from the use of PySpark by developers accustomed to developing with pandas as a data wrangling module. For this reason, they have introduced **Koalas** (<https://koalas.readthedocs.io>), which provides a pandas API on Apache Spark.

The fact remains that Dask has many advantages over Spark in using a distributed system on your laptop only. For example, Spark is based on a **Java Virtual Machine (JVM)** infrastructure, and therefore requires Java and other components to be installed, while Dask is written in pure Python. In addition, the use of Dask enables a faster transition from your laptop to a cluster on the cloud, which can be easily allocated, for example, thanks to Azure. This is made possible thanks to the **Dask Cloud Provider** package (<https://cloudprovider.dask.org/>), which provides classes to create and manage temporary Dask clusters on various cloud platforms. Take a look at the references if you need to create a Dask cluster on Azure via Azure Spot Virtual Machines, or by leveraging Azure Machine Learning compute clusters (using, for example, NVIDIA RAPIDS for GPU-accelerated data science).

Coming back to the topic at hand, let's then see how to install Dask on your laptop.

Installing Dask on your laptop

You will install Dask on the `pbi_powerquery_env` environment, where the pandas and NumPy libraries are already installed. This time, it is not enough to simply run the `pip install dask` command, because this way you'll install only core parts of Dask. The correct way for Dask users is to install all components. In order to display the graph of the execution plan of a Dask operation, a **Graphviz** module must also be installed. To do all this, proceed as follows:

1. Open your Anaconda prompt.
2. Switch to your `pbi_powerquery_env` environment, entering this command:

```
conda activate pbi_powerquery_env
```

3. Enter the following command to install all components of Dask:

```
pip install "dask[complete]"
```

4. Enter the following command to install all components of Graphviz:

```
pip install graphviz
```

You also need to install Graphviz executables in Windows:

1. Go to <http://www.graphviz.org/download/>, and then download and install the stable Windows install package.
2. During the installation, choose to add Graphviz to the system path for the current user.

Let's explore at this point the structures made available by Dask that allow you to extend common interfaces, such as those of NumPy, pandas, and Python iterators, to handle objects larger than the available memory.

Creating a Dask DataFrame

A **Dask DataFrame** is part of the Dask *Big Data* collections that allow pandas, NumPy, and Python iterators to scale easily. In addition to Dask DataFrames, which are the counterpart of pandas DataFrames, **Dask Array** (which mimics NumPy), **Dask Bag** (which mimics iterators), and **Dask Delayed** (which mimics loops) are also part of the collections. However, we will focus on Dask DataFrames, which will allow us to achieve the analysis goal set at the beginning of the chapter.

A Dask DataFrame is nothing more than a set of pandas DataFrames, which can reside on disk on a single machine or on multiple nodes in a cluster, allowing you to manage datasets larger than the RAM on your laptop. We assume that you have already unzipped the CSV files containing data on US flights from 1987 to 2012, as mentioned at the beginning of this chapter, in the `D:\<your-path>\AirOnTimeCSV` folder.

Important Note

If you don't have a laptop with enough hardware resources (at least 16 GB of RAM), you should import a subset of CSV files first (such as 40–50 files) to test the scripts without having to wait excessively long execution times or incurring memory errors.

Then, you can create your Dask DataFrame very easily in the following way:

```
import os
import dask.dataframe as dd
main_path = os.path.join('D:\\', 'your-path', 'AirOnTimeCSV')
ddf = dd.read_csv(
    os.path.join(main_path, 'airOT*.csv'),
    encoding='latin-1',
    usecols=['YEAR', 'MONTH', 'DAY_OF_MONTH', 'ORIGIN', 'DEP_DELAY']
)
```

Note that the wildcard `*` character allows the capture of all CSV files contained in the folder that are in the form `airOTyyyymm.csv`, where `yyyy` indicates the year and `mm` the month number of the flight departure date. Moreover, the encoding of CSV files is declared as `latin-1`.

Important Note

Nowhere is it indicated that the downloaded CSV files had such encoding. By simply trying to import them without declaring it (and therefore assuming `utf-8` by default), loading returns the following strange error:

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe4 in position 4: invalid continuation byte
. Searching the web, it is easy to find that this kind of error is encoding-related and that latin-1 is the correct one.
```

Also, it is a good idea to *specify only the columns of interest* via the `usecols` parameter in order to limit the columns to be read. This practice also guarantees to read only those columns that you are sure are not completely empty, thus avoiding reading errors due to the different inferred data type compared to the real one.

Important Note

It may occur that some columns have a number of null values at the beginning and therefore Dask cannot impute the correct data type, as it uses a sample to do that. In this case, you should specifically declare the data type of those columns by using the `dtype` parameter.

Now that the Dask DataFrame has been created, let's see how to use it to extract the information we need.

Extracting information from a Dask DataFrame

If you have run the code to read all CSV files, you will have noticed that the operation took very little time. Come to think of it, it's really too little time to read 30 GB of data. Could it be that the reading was not successful? The secret of most parallel computing frameworks is tied to this very feature: the read operation has not been physically performed but has been added to a possible queue of operations to be performed when you explicitly request to use the data. This concept is known as **lazy evaluation** or **delayed computation**.

Then, your Dask DataFrame can be used in subsequent operations as if it already contains the data. In our case, as the average airline delay at each US airport for each day of the year is needed, consider using the following code:

```
mean_dep_delay_ddf = ddf.groupby(['YEAR', 'MONTH', 'DAY_OF_MONTH', 'ORIGIN'])[['DEP_DELAY']].mean
```

If you are a little bit familiar with pandas DataFrame transformations, you will notice the use of the same methods for the Dask DataFrame. As with pandas, you have to use *double square brackets* to output a DataFrame; otherwise, you would get a series with a single pair of brackets (take a look here: <http://bit.ly/pandas-subset-df>). Moreover, in order to use the indexes created by the `groupby()` method as columns in a DataFrame, you need to reset them in the `reset_index()` one (for more details, go here: <http://bit.ly/pandas-groupby>).

Executing this piece of code also takes very little time. As you can imagine, the averaging operation has been queued after the data reading operation in the transformation queue, which in this case is assigned to the `mean_dep_delay_ddf` DataFrame. If you want to get a better idea of what the execution plan of the transformations queued so far is, you can create a graph to represent it. For simplicity, we will implement the graph using only one CSV file as input. These are the necessary steps:

1. Create a folder named `AirOnTimeCSVplot`.
2. Copy only the first CSV file you unzipped earlier into the previous folder.
3. Open a new Python script and run the following code on Visual Studio Code:

```
import os
import dask.dataframe as dd
main_path = r'C:\<your-path>\AirOnTimeCSVplot'
ddf_1_month = dd.read_csv(
    os.path.join(main_path, 'airOT*.csv'),
    encoding='latin-1',
    usecols=['YEAR', 'MONTH', 'DAY_OF_MONTH', 'ORIGIN', 'DEP_DELAY']
)
mean_dep_delay_1_month_ddf = ddf_1_month.groupby(['YEAR', 'MONTH', 'DAY_OF_MONTH', 'ORIGIN'])
mean_dep_delay_1_month_ddf.visualize(filename='mean_dep_delay_1_month_dask.pdf')
```

The `visualize()` method allows you to visualize the graph of the tasks estimated by the engine to realize the queued transformations, even before their execution. Specifically, the code will generate a PDF file in the same folder where the script you ran is located.

Starting at the bottom of *Figure 8.1*, which represents the contents of the newly generated PDF file, you can see that the single source CSV file is read by the `read-csv` function from two chunks split by the engine. The `dataframe-groupby-count-chunk` and `dataframe-groupby-sum-chunk` functions are applied to each chunk, since for each tuple defined by the keys of the grouping operation (`YEAR`, `MONTH`, `DAY_OF_MONTH`, and `ORIGIN`), we need to know the sum of the delay (`DEP_DELAY`) and the count of the occurrences to compute the average. After that, the results of the two `dataframe-groupby-sum-chunk` operations on the two chunks are aggregated by the `dataframe-groupby-sum-agg` function. Similarly, the `dataframe-groupby-count-agg` function aggregates the outputs of the two `dataframe-groupby-count-chunk` operations. Once the two DataFrames of sums and counts have been determined, the ratio between the two (that is, the mean) is calculated for each grouping using the `truediv` function. Finally, the `reset_index` function provides the desired DataFrame, the result of the distributed averaging operation.

If you think about it, the famous problem-solving strategy called **Divide and Conquer** (also known as **Divide et Impera**) has been adopted. It consists of dividing the original problem into smaller and generally simpler subproblems, each solved recursively. The solutions of the subproblems are then properly combined to obtain the solution of the original problem. If you've had any experience with the Hadoop world, the **MapReduce** paradigm follows the same philosophy, which was maintained and optimized later by Spark.

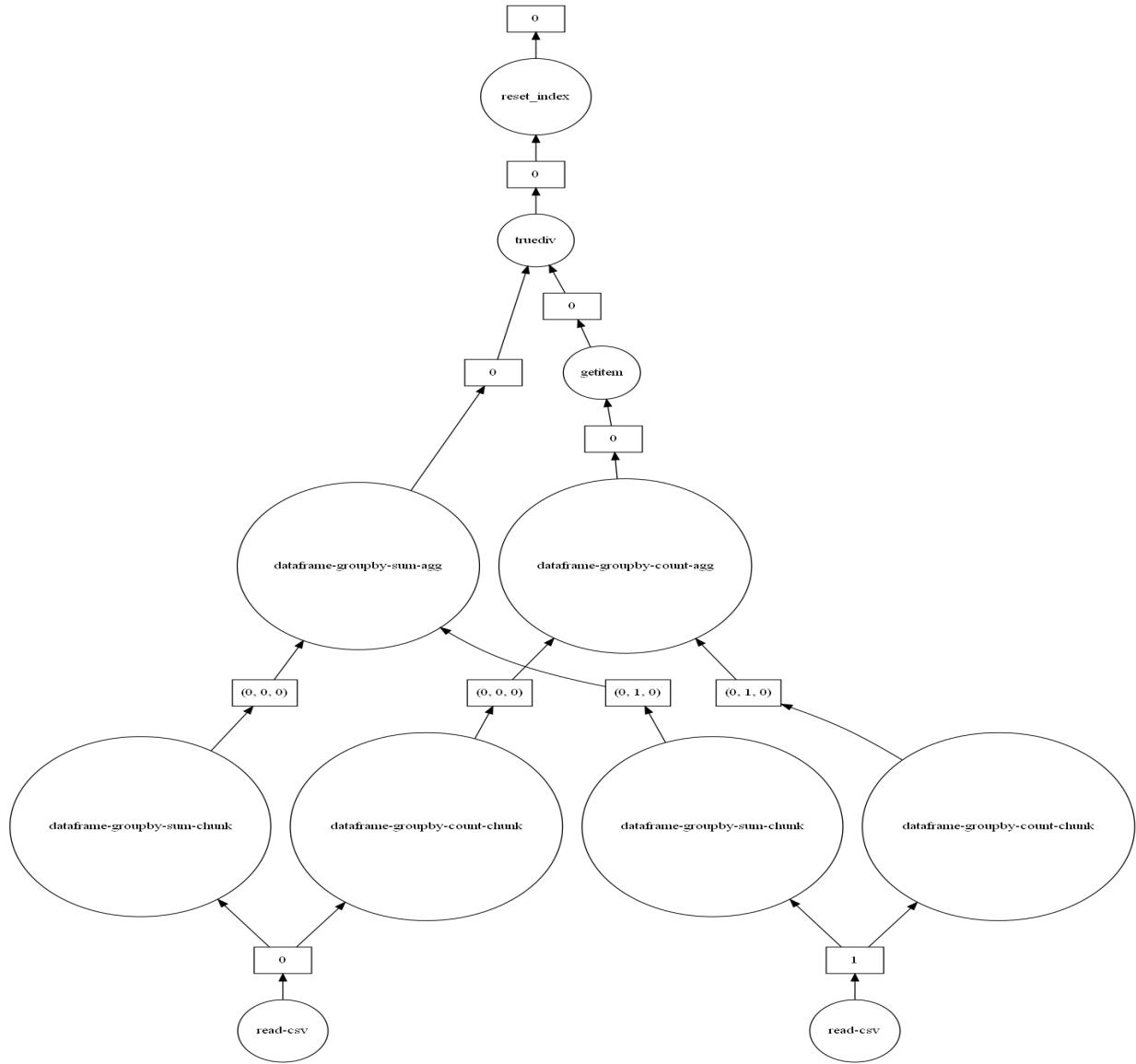


Figure 8.1 – Computation underlying task graph

That said, let's get back to the initial script. We had defined the `mean_dep_delay_ddf` Dask DataFrame. All the transformations needed to get the desired result have been queued. How do you tell Dask to actually proceed with all the computations? You have to explicitly ask for the result via the `compute()` method.

Important Note

Pay attention to the fact that `compute()` returns *in-memory results*. It converts Dask DataFrames to pandas DataFrames, Dask arrays to NumPy arrays, and Dask bags to lists. It is a method that should only be invoked when you are certain that the result will fit comfortably in your machine's RAM. Otherwise, you can write your data on disk using specific methods, such as `to_textfiles()` or `to_parquet()`.

If you instantiated a cluster, you could have decided to persist the calculated data not on disk but in memory, using the `persist()` method. The result would still have been a distributed Dask object but one that references pre-computed results distributed over the cluster's memory.

To add some magic to your script, you can use the `ProgressBar()` object, which allows you to monitor the progress of your computations. Unfortunately, it happens that even if the bar reaches 100%, it still takes some time for the workers to finish processing:

```
[7]▶ with ProgressBar():...  
[  ] [ ##### ] | 100% Completed | 3min 43.7s
```

Figure 8.2 – The progress bar on Visual Studio Code

So, don't give up! Before running the following line of code, keep in mind that processing takes approximately 10 minutes on a machine with 32 GB of RAM and a processor with 6 cores. The script is the following:

```
with ProgressBar():  
    mean_dep_delay_df = mean_dep_delay_ddf.compute()
```

Your laptop will commit all available logical processors to parallelize computations, which you will see in the **Logical Processors** view in **Task Manager**:

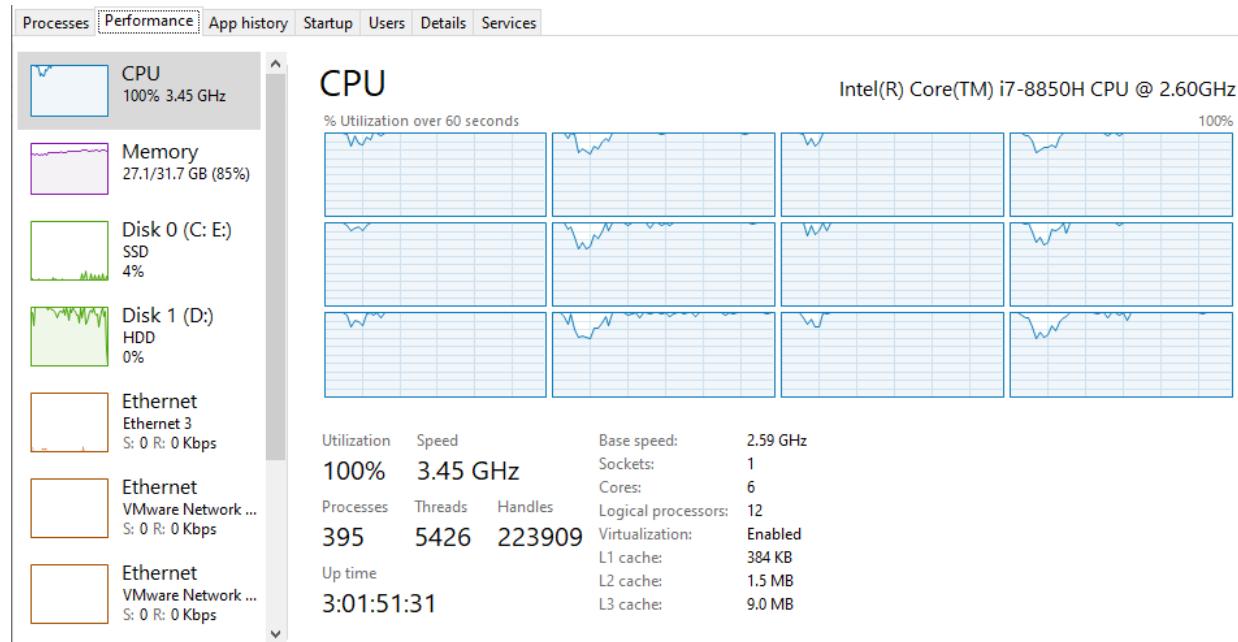


Figure 8.3 – Parallel computations shown in Task Manager

When processing is complete, your pandas DataFrame will be available in memory, and you can view some rows of it as follows:

```
mean_dep_delay_df.head(10)
```

The output will be the following:

[8] mean_dep_delay_df.head(10)

	YEAR	MONTH	DAY_OF_MONTH	ORIGIN	DEP_DELAY
0	1987	10	1	ABE	1.600000
1	1987	10	1	ABQ	2.494253
2	1987	10	1	AGS	2.000000
3	1987	10	1	ALB	5.843750
4	1987	10	1	AMA	0.086957
5	1987	10	1	ANC	1.142857
6	1987	10	1	APF	0.000000
7	1987	10	1	ATL	4.051604
8	1987	10	1	ATW	0.833333
9	1987	10	1	AUS	2.341463

Figure 8.4 – First 10 rows of the output pandas DataFrame

You can find the complete script to create a Dask DataFrame and extract information from it in the `01-load-large-dataset-in-python.py` file in the `Chapter08\Python` folder.

Finally, you were able to get the dataset of a few thousand rows on average flight delays for each airport of origin and for each day of the year by processing a CSV set as large as 30 GB!

If you want to write the contents of a Dask DataFrame to a CSV file, without going through the generation of a pandas DataFrame, you can invoke directly the `to_csv()` method, passing the path of the file to generate as a parameter, as the following example shows:

```
ddf.to_csv(r'D\<your-path>\mean_dep_delay_df.csv')
```

As you can well imagine, calling the `to_csv()` method triggers the actual execution of all queued transformations, just as it did with the `compute()` method, since you're forcing Dask to read the DataFrame content in order to write it to disk. For this reason, if you need to generate the CSV file and also create the pandas DataFrame to be used later in your code, you should not call first `to_csv()` from the Dask DataFrame and then `compute()` to get the pandas DataFrame, because you would force the actual execution of the transformations

pipeline twice. In this case, it is convenient to first generate the pandas DataFrame with `compute()` and then generate a CSV or an Excel file from it, as you learned in *Chapter 7, Logging Data from Power BI to External Sources*.

Let's now try to apply what you have learned so far in Power BI.

Importing a large dataset in Power BI with Python

You learned that Power BI can import data using a Python script directly and that the data must be organized in a pandas DataFrame in order to be used in the data model. Therefore, what you are going to do is develop a script that uses the objects illustrated in the previous section in order to instantiate a pandas DataFrame containing the data of flight delay averages in the US. From this DataFrame, you will then generate a CSV file. Here are the steps required to implement it in Power BI:

1. Open Power BI Desktop and make sure it is referencing your Python `pbi_powerquery_env` environment.
2. Click on **Get data**, start entering `Python`, and then double-click on **Python script**.
3. Copy the content of the `02-load-large-dataset-in-power-bi.py` file into the `Chapter08/Python` folder and paste the content into the Python script editor. Remember to edit the path to the CSV files (`main_path`) and destination path of the CSV file to be generated, and then click **OK**.
4. After about 11 minutes (using a 6-core laptop with 32 GB of RAM), you will see the **Navigator** window showing the aggregated data in the `mean_dep_delay_df` DataFrame:

Navigator

The screenshot shows the Power BI Navigator interface. On the left, there is a tree view under 'Display Options' with 'Python [1]' expanded, showing 'mean_dep_delay_df' selected. On the right, the 'mean_dep_delay_df' DataFrame is displayed in a grid format with the following data:

YEAR	MONTH	DAY_OF_MONTH	ORIGIN	DEP_DELAY
1987	10	1	ABE	1.6
1987	10	1	ABQ	2.494252874
1987	10	1	AGS	2
1987	10	1	ALB	5.84375

Figure 8.5 – The `mean_dep_delay_df` DataFrame loaded in Power BI

Select the DataFrame and click on **Load**. After that, a **Load** popup will appear, and it will remain active for about five more minutes:

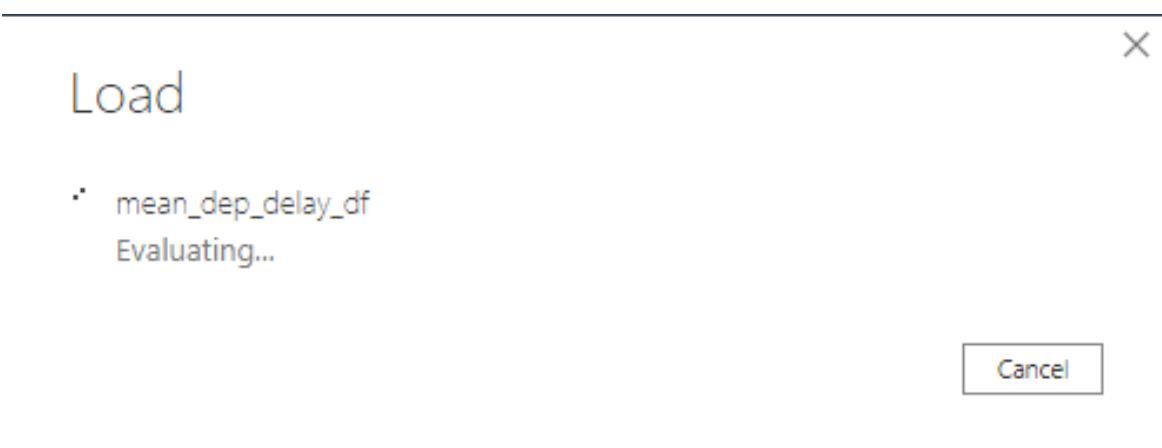
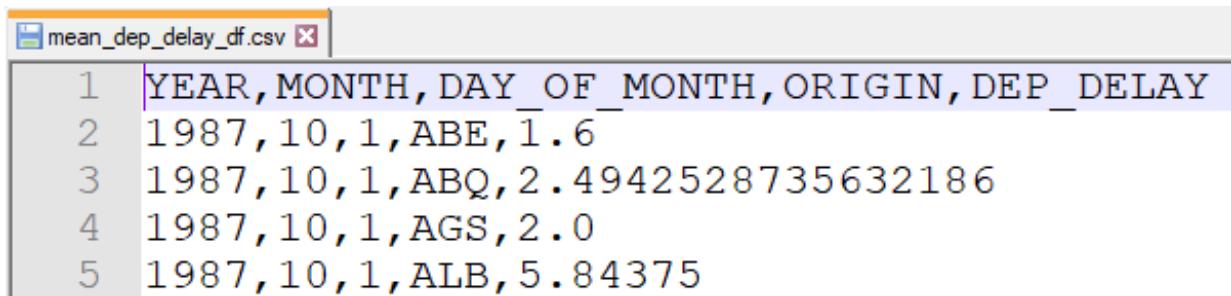


Figure 8.6 – Final phase of loading data into Power BI

When it disappears, your data is loaded into the data model. Moreover, the script also generated the CSV file containing your aggregated data. You'll find the `mean_dep_delay_df.csv` file, sized about 60 MB, in your folder:



	YEAR	MONTH	DAY_OF_MONTH	ORIGIN	DEP_DELAY
1	1987	10	1	ABE	1.6
2	1987	10	1	ABQ	2.4942528735632186
3	1987	10	1	AGS	2.0
4	1987	10	1	ALB	5.84375

Figure 8.7 – Content of the CSV file created from the Python script in Power BI

Impressive! You just imported 30 GB of data to your Power BI Desktop, with just a few lines in Python. Congratulations!

Did you know you can also do this in R? Let's see how.

Importing large datasets with R

The same scalability limitations illustrated for Python packages used to manipulate data also exist for R packages in the **Tidyverse** ecosystem. Even in R, it is not possible to use a dataset larger than the available RAM on the machine. The first solution that is adopted in these cases is also to switch to Spark-based distributed systems, which provide the **SparkR** language. It provides a distributed implementation of the DataFrame you are used to in R, supporting filtering, aggregation, and selection operations as you do with the `dplyr` package. For those of us who are fans of the Tidyverse world, RStudio actively develops the `sparklyr` package, which allows you to use all the functionality of `dplyr`, even for distributed DataFrames. However, adopting Spark-based systems to process CSVs that together take up little more than the RAM you have available on your machine may be overkill because of the overhead introduced by all the Java infrastructure needed to run them.

It is in these cases that adopting the `disk.frame` package (<https://github.com/xiaodaigh/disk.frame>) is the winning approach. What `disk.frame` allows you to do is to divide a dataset larger than RAM into chunks and store each chunk in a separate file within a folder. **Chunk files** are serializations of DataFrames smaller than the initial one in the compressed `fst` format, introduced by the `fst` package (<https://github.com/fstpackage/fst>). It is specifically designed to unleash the *speed and parallelism of solid-state disks* that are easily found in modern computers. DataFrames stored in the `fst` format have full random access, both column and row, and are therefore able to benefit from parallel computations.

Important Note

If you want to benefit from the parallelism introduced by the `disk.frame` package, you must necessarily persist the source CSV files on an SSD disk. If you don't have it, and you use an HDD disk, your processing will take too long, and it won't be worthwhile.

You can define the degree of parallelism with which `disk.frame` will work by indicating the number of parallel workers you want to use. For each worker, it will create an **R session** using the parallelism mechanism provided by the `future` package.

Let's see how to use `disk.frame` in detail.

Installing disk.frame on your laptop

First, you need to install the `disk.frame` package in your R engine:

1. Open RStudio and make sure it is referencing your latest CRAN (Comprehensive R Archive Network) R engine (in our case, version 4.0.2).
2. Click on the console prompt and enter this command: `install.packages("disk.frame")`. Make sure you install the latest version of the package (in our case, version 0.5.0).

At this point, you are ready to test it on RStudio.

Creating a disk.frame instance

In order to create a `disk.frame` instance (a disk-based DataFrame), you must first define the number of parallel workers you want to initialize. Obviously, the higher the number of workers, the faster the `disk.frame` instance will be created. But you cannot initialize any large number of workers, because the number of parallel threads you can use depends on the processor of your machine. Also, you can define a maximum limit to the amount of data you can exchange from one worker to another. For this reason, at the beginning of the script that uses `disk.frame`, you will find the following:

```
library(dplyr)
library(disk.frame)
n_cores <- future::availableCores() - 1
setup_disk.frame(workers = n_cores)
options(future.globals.maxSize = Inf)
```

The `availableCores()` function calculates the number of logical processors available on your machine. Usually, it's best practice to leave one out for computationally demanding tasks, so that the machine doesn't become unresponsive. The `setup_disk.frame()` function creates the cluster of workers needed for the computations. Furthermore, there is no limit to the amount of data that can be exchanged between workers (`Inf` stands for **infinite**).

We assume that you have already unzipped the CSV files containing data on US flights from 1987 to 2012, as mentioned at the beginning of this chapter, in the `D:\<your-path>\AirOnTimeCSV` folder.

Important Note

If you don't have a laptop with enough hardware resources, you should import a subset of CSV files first (such as 40–50 files) to test the scripts without having to wait excessively long execution times.

You have to define a list of paths for each CSV file using the `list.files()` function, and then you can proceed with the creation of `disk.frame` using the `csv_to_disk.frame()` function:

```
main_path <- 'D:/<your-path>/AirOnTime'
air_files <- list.files( paste0(main_path, '/AirOnTimeCSV'), full.names=TRUE )
start_time <- Sys.time()
dkf <- csv_to_disk.frame(
  infile = air_files,
  outdir = paste0(main_path, '/AirOnTime.df'),
  select = c('YEAR', 'MONTH', 'DAY_OF_MONTH', 'ORIGIN', 'DEP_DELAY')
)
end_time <- Sys.time()
(create_dkf_exec_time <- end_time - start_time)
```

The creation of a `disk.frame` instance consists of two steps:

1. The process creates as many chunks as there are CSV files for each allocated worker in temporary folders.
2. Chunks are aggregated and persisted in the output folder (defined by the `outdir` parameter) associated with `disk.frame`.

In short, `disk.frame` is a folder, preferably with a `.df` extension, that contains the aggregated chunks (`.fst` files) generated from the source files. As you've already seen with Dask, it is a good idea to *specify only the columns of interest* via the `select` parameter in order to limit the columns to be read. This practice also

guarantees to read only those columns you are sure are not completely empty, thus avoiding reading errors due to the different inferred data type compared to the real one.

Important Note

It may occur that some columns have a number of null values at the beginning and therefore `disk.frame` cannot impute the correct data type, as it uses a sample to do that. In this case, you should specifically declare the data type of those columns using the `colClasses` parameter.

Note that once a value is assigned to the `create_dkf_exec_time` variable, the round brackets print it on the screen, all in one line. The creation time is about 1 minute and 17 seconds on a machine with 6 cores and 32 GB, using 11 workers.

Once the creation of `disk.frame` is complete, the multiple R sessions that make up the cluster could remain active. In order to force their disconnection, it is useful to invoke the following command:

```
future:::clusterRegistry("stop")
```

In this way, the machine's resources are completely released.

Important Note

A correctly created `disk.frame` can always be referenced later using the `disk.frame()` function by passing it to the path of the `.df` folder, without having to create it again.

You can find the complete script to create `disk.frame` in the `01-create-diskframe-in-r.R` file into the `Chapter08\R` folder.

Now let's see how to get information from the newly created `disk.frame`.

Extracting information from disk.frame

As we are interested in getting the average airline delay at each US airport for each day of the year, a grouping operation is required on the entire `disk.frame`. In order to optimize the execution time, in this case it is also necessary to instantiate a cluster of workers that will read in parallel the information from the newly created `disk.frame`. You can transform the `disk.frame` instance using the syntax you know from the `dplyr` package. Moreover, in order to optimize the machine resources, you will force the engine to read only the columns strictly necessary to the requested computation using the `srckeep()` function from the `disk.frame` package. In our case, it would be unnecessary to select the columns, since when creating the `disk.frame` instance we kept only those strictly necessary, but the following recommendation applies.

Important Note

It's good practice to always use the `srckeep()` function in aggregate data extraction scripts, because if the `disk.frame` instance had all the initial columns, the operation would cause a machine crash.

The following code should be used to extract aggregated data:

```
library(dplyr)
library(disk.frame)
n_cores <- future::availableCores() - 1
setup_disk.frame(workers = n_cores)
options(future.globals.maxSize = Inf)
main_path <- 'D:/<your-path>/AirOnTime'
dkf <- disk.frame( paste0(main_path, '/AirOnTime.df') )
start_time <- Sys.time()
mean_dep_delay_df <- dkf %>%
  srckeep(c("YEAR", "MONTH", "DAY_OF_MONTH", "ORIGIN", "DEP_DELAY")) %>%
  group_by(YEAR, MONTH, DAY_OF_MONTH, ORIGIN) %>%
  summarise(avg_delay = mean(DEP_DELAY, na.rm = TRUE)) %>%
  collect()
```

```

end_time <- Sys.time()
aggregate_exec_time <- end_time - start_time
future:::ClusterRegistry("stop")

```

Again, as already seen for Dask in Python, there are functions that collect all the queued transformations and trigger the execution of calculations. In the case of `disk.frame`, the function is `collect()`. The duration of this operation is about 20 minutes with a 32 GB machine with 6 cores, using 11 workers.

The end result is a tibble containing the desired air delay averages:

```

> head(mean_dep_delay_df, 10)
# A tibble: 10 x 7
# Groups:   YEAR, MONTH, DAY_OF_MONTH [1]
  YEAR MONTH DAY_OF_MONTH ORIGIN avg_delay
  <int> <int>      <int> <chr>    <dbl>
1 1987     10          1 ABE     1.6
2 1987     10          1 ABQ     2.49
3 1987     10          1 ACV     14
4 1987     10          1 AGS      2
5 1987     10          1 ALB     5.84
6 1987     10          1 ALO      0
7 1987     10          1 AMA     0.0870
8 1987     10          1 ANC     1.14
9 1987     10          1 APF      0
10 1987     10         1 ATL     4.02
>

```

Figure 8.8 – First rows of the tibble containing the delay averages

Also in this case, you were able to get the dataset of a few thousand rows on average flight delays by processing a CSV set as large as 30 GB!

You can find the complete script to extract aggregated data from `disk.frame` in the `02-extract-info-from-diskframe-in-r.R` file in the `Chapter08\R` folder.

The following common-sense observation applies here as well.

Important Note

If the results of previous time-consuming processing are to be reused often, it is best to *persist them on disk in a reusable format*. In this way, it will be avoided you can avoid to executing again all the onerous processing of the queued transformations.

A `disk.frame` instance doesn't have direct methods, like a Dask DataFrame, that allow its contents to be written to disk downstream of all queued transformations. Since the result is a tibble, which for all intents and purposes is

a DataFrame, you can write it to disk following the directions you learned in *Chapter 7, Logging Data from Power BI to External Sources*.

So, let's see how you can apply what you've learned so far in Power BI.

Importing a large dataset in Power BI with R

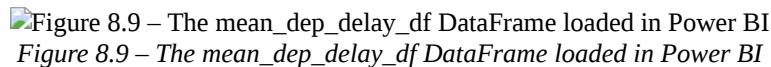
The optimal solution to load a dataset larger than the available RAM with R via Power BI would be to be able to create the `disk.frame` instance directly via an R script in Power BI itself.

Important Note

Unfortunately, the operation of creating a `disk.frame` instance from various CSV files *triggers an error* in Power BI during the second phase of the process (the aggregation of various chunks into the `disk.frame` folder).

Therefore, in order to extract information from a `disk.frame` instance in Power BI, you must first create it in RStudio (or any other IDE or any automated script). Once the `disk.frame` instance has been created, it is then possible to use it in Power BI as follows:

1. Make sure you have correctly created your `disk.frame` instance using RStudio by running the code contained in the `01-create-diskframe-in-r.R` file in the `Chapter08/R` folder.
2. Open Power BI Desktop and make sure it is referencing your latest CRAN R engine.
3. Click on **Get data**, start entering `script`, and then double-click on **R script**.
4. Copy the content of the `03-extract-info-from-diskframe-in-power-bi.R` file into the `Chapter08\R` folder and paste it into the R script editor. Remember to edit the destination path of the CSV file to be generated, and then click **OK**.
5. After about 30 minutes (using a 6-core laptop with 32 GB of RAM), you will see the **Navigator** window showing the aggregated data in the `mean_dep_delay_df` DataFrame:

Figure 8.9 – The `mean_dep_delay_df` DataFrame loaded in Power BI
Figure 8.9 – The `mean_dep_delay_df` DataFrame loaded in Power BI

Select the DataFrame and click on **Load**. After that, a **Load** popup will appear, and it will remain active for about 10 more minutes:

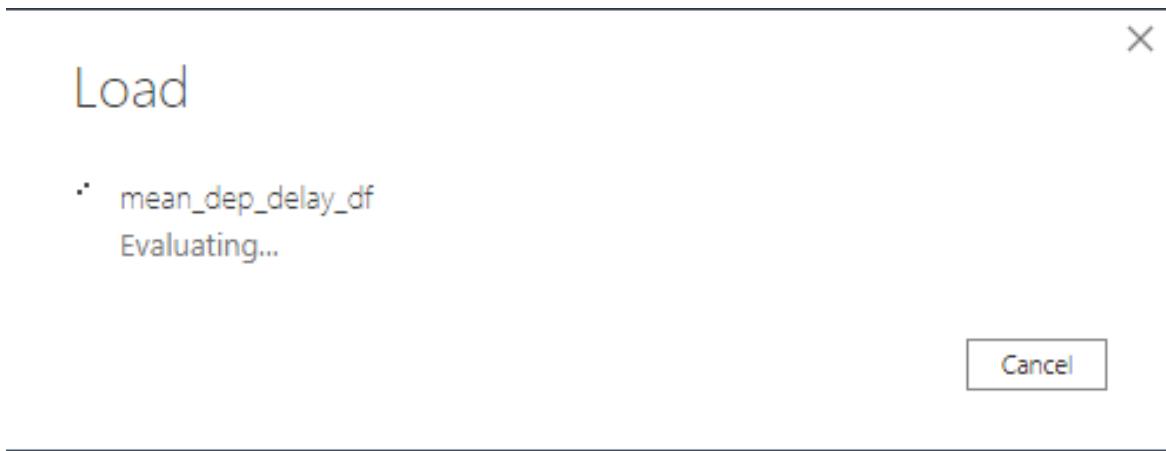


Figure 8.10 – Final phase of loading data into Power BI

When it disappears, your data is loaded into the data model. Moreover, the script also generated the CSV file containing your aggregated data. You'll find the `mean_dep_delay_df.csv` file, sized about 60 MB, in your

folder:

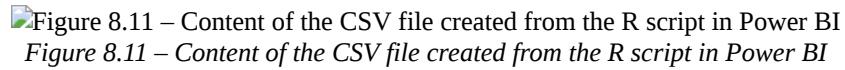


Figure 8.11 – Content of the CSV file created from the R script in Power BI
Figure 8.11 – Content of the CSV file created from the R script in Power BI

Awesome! Would you ever have thought of importing 30 GB of data into your Power BI Desktop to extract information from it in R? Well, you just did it with just a few lines of code.

Summary

In this chapter, you learned how to import datasets larger than the RAM available to your laptop into Python and R. You've also applied this knowledge to import a set of CSV files that, in total, take up more than the available RAM on your machine into Power BI Desktop.

In the next chapter, you will learn how to enrich your data with external information extracted from web services made available to users.

References

For additional reading, check out the following books and articles:

1. *Create Dask clusters on Azure using Azure (Spot) VMs* (<https://cloudprovider.dask.org/en/latest/azure.html>)
2. *Accelerated Machine Learning at Scale with NVIDIA RAPIDS on Microsoft Azure ML with Dask* (https://github.com/drabastomek/GTC/tree/master/SJ_2020/workshop)

9 Calling External APIs to Enrich Your Data

In the previous chapter, you saw an example of how to enrich the data you already have with external information. In that instance, the data was provided via CSV files, but this is not always the case. Very often, the data useful for enrichment is exposed via external **Application Programming Interfaces (APIs)**, most frequently in the form of web service endpoints. Power BI allows you to read data from a web service via a dedicated UI, but most of the time it is unusable. Therefore, you have to resort to writing **M code** to get it done. Writing M code isn't too difficult, but it's not that straightforward either. You also have to be careful not to write code that leads to refresh issues when publishing the report to the Power BI service. Moreover, in Power BI, it is not possible to parallelize more than one call to the same web service in order to reduce waiting time when retrieving data. Adopting Python or R to get data from a web service solves all these issues very easily.

In this chapter, you will learn the following topics:

- What a web service is
- Registering for Bing Maps Web Services
- Geocoding addresses using Python
- Geocoding addresses using R
- Accessing web services using Power BI

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

What a web service is

In the course of your work as an analyst, you may need to retrieve data through an API, exposed by a system within your network, for example. However, this is a rare case, since today almost all external data sources are exposed as **web services**, even within a company.

Web services are the most common and popular way of communicating information between heterogeneous information systems. A web service is basically a software module hosted on a server that is available over the internet to provide data to specific requests from a client.

There are mostly two types of design models for web services: **Simple Object Access Protocol (SOAP)** and **REpresentational State Transfer (REST)**.

- SOAP relies heavily on **XML** and defines a highly typed messaging structure through schemas. All messages exchanged between the service and the client are all encoded through **Web Service Definition Language (WSDL)**, which, in turn, is based on the XML format. One of the most important aspects of WSDL is that it defines a binding contract between the service provider and each service consumer. Therefore, any change to the API involves a change to be made to the client. Nowadays, almost everything that matters runs on HTTP. But keep in mind that, in addition to HTTP, SOAP can take advantage of any other transport protocol (such as SMTP and TCP).
- REST is becoming the default design model for all public APIs. It is an architecture that relies exclusively on the HTTP protocol (as opposed to SOAP). It doesn't use WSDL contracts and is, therefore, more flexible and faster to implement. REST can handle data in any format, such as XML or YAML, but the most used is surely **JSON**. Unlike SOAP, which is function-driven, REST is very **data-driven**. It is for this reason that all web services used for data enrichment are **RESTful** (take a look at the references for more details) and can generate output in any format – not only JSON but also CSV, or **Really Simple Syndication (RSS)**, for

example! Basically, REST offers a lighter method of interacting with the service, using URLs in most cases to receive or send information. The exposed basic methods are `GET`, `POST`, `PUT`, and `DELETE`.

Quite often, you will have heard about web service **endpoints**. In general, a web service works by accepting `GET` requests from the client and providing responses to them. Considering a REST API, an endpoint is a URL at which the web service can be accessed by a client application. A web service can provide more than one endpoint. If you consider **Bing Maps REST Services**, for example, the endpoint used for geocoding is as follows: `dev.virtualearth.net/REST/v1/Locations`. The one used instead to provide a route between two waypoints is this one: `dev.virtualearth.net/REST/v1/Routes`.

Now that it's clear what a web service is and what the meaning of the technical terms that often come with it is, we can move on to showing you how to use a RESTful one.

Registering for Bing Maps Web Services

In this chapter, we will use Bing Maps Web Services as an example. Therefore, you need to create a free Bing Maps Basic Key via your Microsoft account. The necessary steps to do so are as follows:

1. Go to <https://www.microsoft.com/en-us/maps/create-a-bing-maps-key>, select the **Basic Key** option just after the main banner, and then click on **Get a free Basic key** button.
2. On the next **Welcome** page, click on **Sign in** and use your Microsoft Account to log in.
3. On the next page, you will be notified that this is the first time your email is being used to authenticate in the **Bing Maps Dev Center**. Click on **Yes, let's create a new account** to create a new account.
4. Enter your account details on the next page and click **Create**. After that, you'll be logged in to the Dev Center, and you'll see some claims on the page saying **Announcement** and **Important reminder**.
5. On that page, under the **My account** menu, click on **My keys**.
6. You will be prompted with a form asking for a new key. Just fill the required fields, entering `geocoding-test` in the **Application name** field, and leaving `Basic` as **Key type** and `Dev/Test` as **Application type**. Then, click on **Create**.

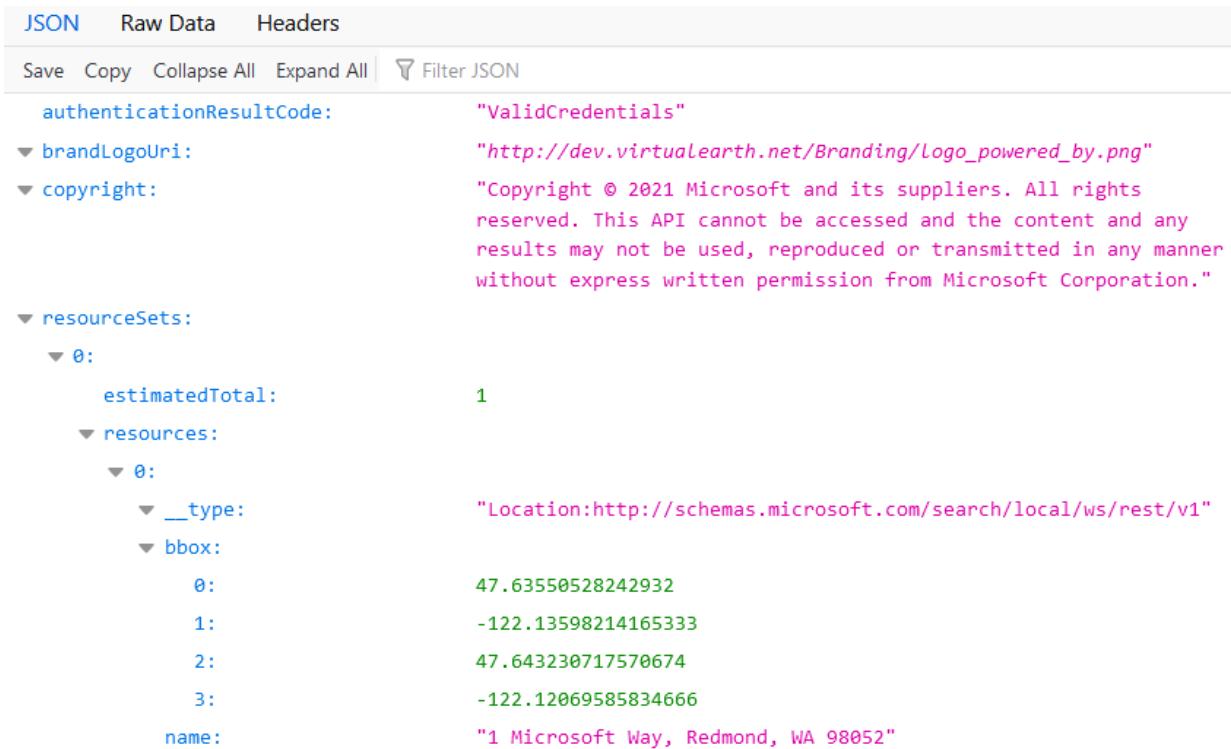
7. A page will appear to confirm that the key has been created, such as the following one:

The screenshot shows the 'My keys' page of the Bing Maps Dev Center. At the top, there are navigation links: 'My account', 'Data sources', 'Announcements', 'Contacts & Info', and 'Hello'. Below the header, a message says 'Key created successfully.' There are two links: 'Click here to create a new key.' and 'Click here to download complete list of keys.' A search bar with placeholder text 'Enter key to search...' and a magnifying glass icon is present. A red arrow points from the text 'Key: Show key' in the table below to the search bar. The table has columns 'Application name' and 'Key details'. The row for the newly created key 'geocoding-test' shows the following details: Key: Show key, Application Url: https://dev.virtualearth.net/REST/v1/Locations, Key type: Basic / Dev/Test, Created date: 05/07/2021, Expiration date: None, Key Status: Enabled, Security Enabled: No. To the right of the table are buttons for 'Update', 'Copy key', 'Usage Report', 'Enable Security', and 'Enable Preview'.

Application name	Key details	Enable Preview for All Keys
geocoding-test	Key: Show key Application Url: https://dev.virtualearth.net/REST/v1/Locations Key type: Basic / Dev/Test Created date: 05/07/2021 Expiration date: None Key Status: Enabled Security Enabled: No	<input checked="" type="checkbox"/> Update <input checked="" type="checkbox"/> Copy key <input checked="" type="checkbox"/> Usage Report <input checked="" type="checkbox"/> Enable Security <input checked="" type="checkbox"/> Enable Preview

Figure 9.1 – Bing Maps key confirmation

8. Click on **Show key** to see the key you'll use in the next examples and copy it.
9. Enter the following URL in your browser URL bar, replacing the <your-bing-maps-api-key> string with your key:
<http://dev.virtualearth.net/REST/v1/Locations/1%20Microsoft%20Way%20Redmond%20WA%2098052?key=<your-bing-maps-api-key>>. Then, press *Enter*.
10. Some browsers, such as Firefox, prettify JSON responses returned by web services. In your case, if all goes well, you should see a result such as the following:



The screenshot shows a JSON viewer interface with tabs for 'JSON', 'Raw Data', and 'Headers'. Below the tabs are buttons for 'Save', 'Copy', 'Collapse All', 'Expand All', and a 'Filter JSON' input field. The main area displays a nested JSON object representing the geocoding response. The structure includes fields like 'authenticationResultCode', 'brandLogoUri', 'copyright', and 'resourceSets'. The first resource set ('0') contains one resource ('0') which has a bounding box ('bbox') defined by coordinates (0, 1), (1, 2), (2, 3), and (3, 4) and a name ('name') of '1 Microsoft Way, Redmond, WA 98052'.

```

{
  "authenticationResultCode": "ValidCredentials",
  "brandLogoUri": "http://dev.virtualearth.net/Branding/logo_powered_by.png",
  "copyright": "Copyright © 2021 Microsoft and its suppliers. All rights reserved. This API cannot be accessed and the content and any results may not be used, reproduced or transmitted in any manner without express written permission from Microsoft Corporation.",
  "resourceSets": [
    {
      "0": {
        "estimatedTotal": 1,
        "resources": [
          {
            "0": {
              "__type": "Location:http://schemas.microsoft.com/search/local/ws/rest/v1",
              "bbox": [
                {
                  "0": 47.63550528242932,
                  "1": -122.13598214165333
                },
                {
                  "2": 47.643230717570674,
                  "3": -122.12069585834666
                }
              ],
              "name": "1 Microsoft Way, Redmond, WA 98052"
            }
          }
        ]
      }
    }
  ]
}

```

Figure 9.2 – Your first geocoding using the Bing Maps Locations API via the browser

Great! You've just geocoded an address passed as a query parameter to the Bing Maps Locations API! Let's now see how to use Python to automate this process.

Geocoding addresses using Python

In this section, we'll show you how to make calls to the Bing Maps Locations API using both a direct call to the URL via the `GET` method (which is ultimately equivalent to the example call you made earlier via the browser) and a dedicated Python **Software Development Kit (SDK)** that facilitates the query.

Using an explicit GET request

If we want to receive geocoding data for an address from the Bing API, we need to make a request to the web service by passing the address of interest as a parameter. The parameters are passed through appropriate concatenation of the parameters with the URL of the endpoint. In our case, the full format of the endpoint URL useful for geocoding an address is as follows:

[base_url]query=[address]?key=[AUTH_KEY]

Figure 9.3 – URL format of a GET request to the Bing Maps Locations API

The following is the definition of each string token you can see in *Figure 9.3*:

- `base_url` : The endpoint URL string, that is, `http://dev.virtualearth.net/REST/v1/Locations/`
- `address` : The string of the address that you want to geocode, transformed using the *percent encoding* technique to avoid using special characters in the final URL
- `AUTH_KEY` : Your Bing key

Once the URL has been built in this way, it can be used to make a `GET` request via the `get()` method of the `requests` module. After getting the data as a result of the request, you'll capture the content of the web service response containing JSON, and from that, you can extract the values of interest. For example, in order to extract the `formattedAddress` value, you have to navigate into the data structure, as shown in *Figure 9.4*:



Figure 9.4 – Visual structure of the Bing Maps Locations API response

So, you can navigate your JSON variable `data`, in the same way to get that value, as follows:

```
data['resourceSets'][0]['resources'][0]['address']['formattedAddress']
```

From *Figure 9.4*, you can see that the `resources` node can contain more than one subnode, identified with an integer. This is because sometimes the geocoding engine cannot exactly identify the geographic point from the passed address and therefore returns more than one result. The `estimatedTotal` attribute indicates the number of possible geocodes identified. For simplicity, we have extracted from the JSON the last identified resource, which is the one with the highest identifier.

Besides the answer in JSON format, from the request, you can also extract other values, such as the reason for the state of the answer (useful to understand whether the `GET` operation is successful or not), the complete content of the answer in text format, and the URL used for the `GET` request. You will need the `requests`, `urllib`, `json`, `pandas`, and `time` modules. The latter is used to measure the geocoding execution time of all addresses in the dataset. Everything can be encapsulated in a handy function called `bing_geocode_via_address()`:

```

def bing_geocode_via_address(address):
    # trim the string from leading and trailing spaces using strip
    full_url = f"{base_url}query={urllib.parse.quote(address.strip(), safe='')}?key={AUTH_KEY}"
    r = requests.get(full_url)
    try:
        data = r.json()
        # number of resources found, used as index to get
        # the latest resource
        num_resources = data['resourceSets'][0]['estimatedTotal']
        formattedAddress = data['resourceSets'][0]['resources'][num_resources-1]['address']['formattedAddress']
        lat = data['resourceSets'][0]['resources'][num_resources-1]['point']['coordinates'][0]
        lng = data['resourceSets'][0]['resources'][num_resources-1]['point']['coordinates'][1]
    except:
        num_resources = 0
        formattedAddress = None
        lat = None
        lng = None

    text = r.text
    status = r.reason
    url = r.url

    return num_resources, formattedAddress, lat, lng, text, status, url

```

The code chunks contained in this section are available in the `01-geocoding-with-python.py` file inside the `Chapter09\Python` folder.

Assuming we have a list of addresses to geocode in a pandas DataFrame, it makes sense to write a handy function that takes as input a DataFrame row and the column name in which the address is stored. It will invoke the previous function to get geocoding values to add to the current row and will return it:

```

def enrich_with_geocoding(passed_row, col_name):
    # Fixed waiting time to avoid the "Too many requests"
    # error as basic accounts are limited to 5 queries per second
    time.sleep(3)
    address_value = str(passed_row[col_name])

    num_resources, address_formatted, address_lat, address_lng, text, status, url = bing_geocode_via_address(address_value)

    passed_row['numResources'] = num_resources
    passed_row['formattedAddress'] = address_formatted
    passed_row['latitude'] = address_lat
    passed_row['longitude'] = address_lng
    passed_row['text'] = text
    passed_row['status'] = status
    passed_row['url'] = url

    return passed_row

```

You can test these functions using the test dataset at this link: <http://bit.ly/geocoding-test-addresses>. It's also available in the `Chapter09` folder.

You have to simply pass the previous function as a parameter to the addresses' DataFrame `apply()` method to apply it to each of its rows, shown as follows:

```

base_url= "http://dev.virtualearth.net/REST/v1/Locations/"
AUTH_KEY = os.environ.get('BINGMAPS_API_KEY')
df_orig = pd.read_csv(r'D:\<your-path>\Chapter09\geocoding_test_data.csv', encoding='latin-1')
df = df_orig[['full_address','lat_true','lon_true']]
tic = time.perf_counter()
enriched_df = df.apply(enrich_with_geocoding, col_name='full_address', axis=1)
toc = time.perf_counter()
print(f"{df.shape[0]} addresses geocoded in {toc - tic:0.4f} seconds")

```

Note that the Bing Maps services key is set using the `os.environ.get('BINGMAPS_API_KEY')` function call. This way of accessing sensitive data avoids having to write it in plain text in the code. So, it must have been previously written to the `BINGMAPS_API_KEY` environment variable. You can also do this with Python via the following script:

```
os.environ['BINGMAPS_API_KEY'] = '<your-bing-api-key>'
```

This way, however, each time you restart Visual Studio Code, that variable will be lost. To have it permanently available, you must set up a user variable environment directly through your operating system. In Windows, you can follow this guide: <https://phoenixnap.com/kb/windows-set-environment-variable>. Once added, you must restart Visual Studio Code to make it visible to your code.

Important Note

We preferred to load the CSV file directly into the Python script. It would have been equivalent to first loading the CSV file via the Power BI **Text/CSV** connector and then adding a transformation step that would run the Python code we just analyzed.

The geocoding operation takes about 34 seconds for 120 addresses. Part of the content of the final DataFrame is as follows:

	full_address	lat_true	lon_true	numResources	formattedAddress	latitude	longitude	text	status	url
0	200 K St NE, Washington DC, 20002	38.903155	-77.003274	1	200 K St NE, Washington, DC 20002	38.903161	-77.003053	{"authent":...	OK	http://dev.virtualearth'...
1	200 K St North East, Washington DC, 20002	38.903155	-77.003274	1	200 K St NE, Washington, DC 20002	38.903161	-77.003053	{"authent":...	OK	http://dev.virtualearth'...
2	200 K St Northeast, DC	38.903155	-77.003274	1	200 K St NE, Washington, DC 20002	38.903161	-77.003053	{"authent":...	OK	http://dev.virtualearth'...

Figure 9.5 – Content of the geocoded DataFrame

Impressive! You geocoded as many as 120 addresses with just a few lines of Python code in 34 seconds. But did you know that you can also parallelize `GET` requests by geocoding all your addresses in less time? Let's see how you can achieve this.

Using an explicit GET request in parallel

Just as you learned in *Chapter 8, Loading Large Datasets beyond the Available RAM in Power BI*, the Swiss army knife that allows you to parallelize your computations in Python is Dask. The great thing is that a Dask DataFrame exposes the `apply()` method, which has the same functionality as a pandas DataFrame's `apply()` function, with the difference that it is parallelized. Therefore, the code used in the previous section is practically reusable with a few minor modifications to achieve significantly reduced execution times. You can find the full script in the `02-geocoding-parallel-with-python.py` file in the `Chapter09\Python` folder.

Important Note

When the API requires it, it is preferable to provide multiple inputs using **batch mode**, rather than making multiple calls in parallel. However, if batch mode is not covered by the API, multiple parallel calls are the only way to improve execution times. The Bing Maps Locations API supports batch mode, but for demonstration purposes, we preferred to adopt the multiple call technique.

The `enrich_with_geocoding()` and `bing_geocode_via_address()` functions remain as they are. Instead, the Dask DataFrame is introduced already at the data reading stage, as follows:

```
import dask.dataframe as dd
ddf_orig = dd.read_csv(r'D:\<your-path>\Chapter09\geocoding_test_data.csv', encoding='latin-1')
ddf = ddf_orig[['full_address', 'lat_true', 'lon_true']]
```

Internally, a Dask DataFrame is divided into many partitions, where each partition is a pandas DataFrame. Now, it happens that when the data is not a large amount, it may be loaded in a single partition by `read_csv()`. In fact, if you run the following code: `ddf.npartitions`, you will see that it returns `1`. It is for this reason that in cases

such as this, it is necessary to repartition the Dask DataFrame into an appropriate number of partitions to benefit from parallelism, as follows:

```
ddf = ddf.repartition(npartitions=os.cpu_count()*2)
```

The following applies to the number of partitions in a DataFrame:

Important Note

There is no precise rule for determining the ideal number of partitions in a DataFrame. The formula used earlier is effective for the dataset we are considering here and has been determined empirically.

At this point, you have to simply invoke the `apply()` method of the Dask DataFrame by passing it the `enrich_with_geocoding()` function, as you did with the `apply()` method of the pandas DataFrame in the previous section:

```
enriched_ddf = ddf.apply(enrich_with_geocoding, axis=1, col_name='full_address', meta={'full_address': 'string'})
```

In this case, we had to specify the expected metadata of the output. This is because many operations on DataFrames rely on knowing the name and type of the column. Internally, Dask DataFrame does its best to propagate this information through all operations. Usually, this is done by evaluating the operation on a small sample of dummy data. Sometimes, however, this operation can fail in user-defined functions (such as in the case of `apply`). In these cases, many functions support an optional `meta` keyword, which allows you to specify the metadata directly, avoiding the inference step.

Important Note

Dask DataFrame's `apply()` method now supports only the `axis=1` mode (row-wise).

You've just defined a lazy transformation. To actually perform it, you must use the `compute()` function, so in this case, you can measure the time it takes to geocode all the addresses:

```
tic = time.perf_counter()
enriched_df = enriched_ddf.compute()
toc = time.perf_counter()
print(f'{enriched_df.shape[0]} addresses geocoded in {toc - tic:.4f} seconds')
```

The end result is stunning: 120 addresses were geocoded in *just 3.5 seconds* versus 35 seconds for sequential code! You have achieved a 10x improvement in running time thanks to Dask!

As you've seen in this section and the previous one, the most uncomfortable part of the code is perhaps retrieving every single value from the JSON returned by the web service (the logic in the `bing_geocode_via_address()` function), because you have to know in advance the structure of the result obtained. However, there are Python modules that contain functions that facilitate interaction with specific web services. Let's see how to use the `Geocoder` library, which simplifies the adoption of the most widely used geocoding providers, such as those of Google and Bing.

Using the Geocoder library in parallel

The **Geocoder** module (<https://geocoder.readthedocs.io/>) is a simple and consistent geocoding library written in Python. It makes consistent responses from expected geocoding providers using a unique JSON schema. Some of the providers available are Google, Bing, Mapbox, and TomTom.

First, you need to install the Geocoder library in the `pbi_powerquery_env` environment. You can do that by following these steps:

1. Open the Anaconda prompt.
2. Switch to the `pbi_powerquery_env` environment by entering this command:
`conda activate pbi_powerquery_env`.

3. Install the Geocoder library by entering this command: `pip install geocoder`.

Thanks to Geocoder, the `GET` request for an address is simply transformed into the command `r = geocoder.bing(address, key = AUTH_KEY)`. And the great thing is that the returned object, `r`, already contains the attributes useful for geocoding, such as `r.address`, `r.lat`, or `r.lng`. Therefore, the `bing_geocode_via_address()` function you encountered in the previous sections is embarrassingly simplified, as follows:

```
def bing_geocode_via_address(address):
    r = geocoder.bing(address, key = AUTH_KEY)
    return r.address, r.lat, r.lng, r.json, r.status, r.url
```

In this case, Geocoder selects the most appropriate match if the geocoding operation returns more than one match, saving you from further headaches. The option to return multiple results is currently an in-progress development. In more recent versions of the package, it is possible to pass the `MaxRows` parameter with a value greater than one to the geocoding functions to obtain as many possible results as the value passed. In this way, it is the analyst who can choose the outcome that meets their needs.

You can find the complete script that uses the Geocoder library to geocode addresses in parallel in the `03-geocoding-parallel-using-sdk-with-python.py` file in the `Chapter09\Python` folder.

You can now understand that having an SDK available that is designed to simplify your life in using a specific API is a major boon.

Important Note

One of the overarching principles we always suggest is to *avoid reinventing the wheel* every time you need it. If you need to accomplish a goal (in our case, geocoding an address), most likely someone before you has already thought of it and perhaps shared a library with the community that simplifies your life. Always spend a half-hour searching the web for possible pre-existing solutions that can smartly solve your problem. In addition to saving valuable time, you avoid the risk of going crazy with complexities that others have already overcome for you.

Although we knew of the existence of the Geocoder library, we still wanted to show you how to make `GET` requests to the web service from scratch in the previous sections. This is because it is not certain that SDKs exist for all web services that you may need in the future.

Let's now see how to get the same results using R.

Geocoding addresses using R

You have just learned how to query a web service via raw `GET` requests to the endpoint and via a handy SDK using Python. As you can already guess, you can also do both with R. Let's see how it's done.

Using an explicit GET request

The package that allows calls to URLs in R is `httr` (installed by default with the engine). Making a `GET` request simply translates to `GET(your_url)` thanks to it. As you have already seen in the previous sections, **percent encoding** must be applied to the address to be passed as a web parameter to the Bing Maps Locations API endpoint. The function that allows you to apply this type of encoding to a string is found in the `RCurl` package and is named `curlPercentEncode()`. In addition, the handy `tictoc` package will be used to measure run times. Both the `RCurl` and `tictoc` packages need to be installed, as shown in the following steps:

1. Open RStudio and make sure it is referencing your latest CRAN R engine in **Global Options**.
2. Click on the **Console** window and enter this command: `install.packages("RCurl")`. Then press **Enter** key.
3. Enter this command: `install.packages("tictoc")`. Then press **Enter**.

At this point, once you have defined the `base_url` and `AUTH_KEY` variables specific to the web service to query, it is enough to execute the following code to obtain the result of a `GET` request:

```
encoded_address <- RCurl::curlPercentEncode(address)
full_url <- stringr::str_glue('{base_url}query={encoded_address}?key={AUTH_KEY}')
r <- httr::GET(full_url)
```

The `str_glue()` function of the `stringr` package has been used to concatenate the strings. Note that also, in this case, the `AUTH_KEY` variable is set from an environment variable using `AUTH_KEY = Sys.getenv('BINGMAPS_API_KEY')`. This means that the environment variable must be set beforehand. You can set it directly in your R session using the following R script:

```
Sys.setenv(BINGMAPS_API_KEY = '<your-bing-api-key>')
```

This way, however, each time you restart RStudio, that variable will be lost. To have it permanently available, you must set up a user variable environment directly through your operating system. In Windows, you can follow this guide: <https://phoenixnap.com/kb/windows-set-environment-variable>. Once added, you must restart RStudio to make it visible to your code.

Once you have obtained the result of the request, you can proceed to parse the JSON content to obtain the desired geocoding values. Everything that was taken into account for parsing the result in the previous sections still applies. So again, you will have to handle the multiple results that some geocoding operations might return. As already done for the Python scripts used previously, all this logic can be encapsulated into the `bing_geocode_via_address()` function in R that returns a list of values obtained as a result of geocoding a passed address. You can find the code in the `01-geocoding-with-r.R` file in the `R\Chapter09` folder.

Once you have loaded the contents of the `geocoding_test_data.csv` file into the `tbl_orig` tibble and selected the columns of interest in `tbl`, you will exploit the convenience of the `purrr` package's `map()` function to execute the previous defined `bing_geocode_via_address()` function for each address extracted from the tibble:

```
tic()
tbl_enriched <- tbl %>%
  pull( full_address ) %>%
  map_dfr( ~ bing_geocode_via_address(.x) ) %>%
  bind_cols( tbl, . )
toc()
```

Note how we used the `pull()` function to transform the `full_address` tibble column into a vector and then passed it as a parameter to the `map_dfr()` function, which only accepts lists or vectors as input. You may be wondering what the `map_dfr()` function is for. It is part of the `purrr` family of `map()` functions. While `map()` returns a list of results (each obtained by applying the input function to each element of the input vector) as output, `map_dfr()` directly binds rows when each element of the `map()` output is a named DataFrame, list, or vector. So the final result of `map_dfr()` is a DataFrame/tibble composed of the elements returned by the input function arranged row by row. The whole logic is wrapped by the pair of functions, `tic()` and `toc()`. By running the entire block of code (from `tic()` to `toc()` inclusive), you can get the execution time of the code inside very conveniently.

You may have noticed that there is no need in this case for an intermediate function such as `enrich_with_geocoding()` that accepts the individual lines of the tibble as a parameter. Since the `bing_geocode_via_address()` function returns a named list, the `map_dfr()` function manages to interpret it correctly and bind it into a single tibble.

Important Note

We preferred to load the CSV file directly into the R script. It would have been equivalent to first loading the CSV file via the Power BI **Text/CSV** connector and then adding a transformation step that would run the R code we just analyzed.

Since `map_dfr()` only returns a tibble of geocoding values, you have to bind this tibble to the initial `tbl` one in order to have a single enriched tibble. For this reason, we used the function `bind_cols(tbl, .)`, where the

parameter `.` denotes the tibble of geocoding values passed as a parameter by the piping operation. That said, the whole geocoding operation takes about 30 seconds (which is comparable to the result obtained with Python), and the final tibble will look as follows:

Figure 9.6 – Content of the geocoded DataFrame
Figure 9.6 – Content of the geocoded DataFrame

Well done! You were able to geocode addresses via the web service even with R. Easy, right? Are you now curious to learn how to do this by taking advantage of the parallelism provided by your machine? Let's see how to proceed.

Using an explicit GET request in parallel

Just as in Python, we had to use the Dask module to parallelize computations; in R, we need to introduce a new package to achieve the same thing. The new package is called `furrr` (<https://furrr.futureverse.org/>) and is intended to combine the expressive power of the `purrr` family of mapping functions with the parallel processing capabilities provided by the `future` package (<https://future.futureverse.org/>). Both the `furrr` and `future` packages are part of an interesting framework called **Futureverse** (<https://www.futureverse.org/>), which aims to parallelize existing R code in the simplest way. In practical terms, `furrr` allows you to replace the `map()` and `map_*` functions of `purrr` with the `future_map()` and `future_map_*` functions of `furrr` with minimal effort, and your code will magically run in parallel. Keep in mind that the future engine is also the backend used by the `disk.frame` packages you learned about in *Chapter 8, Loading Large Datasets beyond the Available RAM in Power BI*.

First of all, you need to install the `furrr` package. Just run the command `install.packages("furrr")` in the RStudio's **Console** window and that's it.

To apply what has just been said to the code analyzed in the previous section, it is enough to modify the last part of it as follows to obtain the same result (and obviously loading the `furrr` library instead of `purrr`), but while parallelizing the computations:

```
n_cores <- availableCores() - 1
plan(cluster, workers = n_cores)
tic()
tbl_enriched <- tbl %>%
  pull( full_address ) %>%
  future_map_dfr( ~ bing_geocode_via_address(.x) ) %>%
  bind_cols( tbl, . )
toc()
```

Thanks to the `availableCores()` function, it is possible to identify the number of virtual processors present on the machine. It is good practice not to use all of them as it can make the machine unresponsive. The `plan()` function of a `future` instance allows you to define the strategy with which the future engine performs the calculations (synchronously or asynchronously). It also allows you to define the number of workers that will work in parallel.

Important Note

Generally, the default strategy used on Windows machines is **multisession** and it works fine running the code in RStudio. We found that with this strategy, Power BI cannot handle the multiple sessions generated to parallelize computations. Instead, we found that selecting the **cluster** strategy, despite the machine being unique, allows Power BI to complete the computations.

Another minimal change was to declare the `base_url` and `AUTH` variables associated with the web service directly in the function invoked by `future_map_dfr()` instead of in the main code for simplicity. Passing variables to functions invoked via `furrr` is done slightly differently than in standard practice (follow this link: <https://furrr.futureverse.org/articles/articles/gotchas.html>), and we wanted to avoid adding minimal complexity so as not to distract from the main concept.

You can find the full script in the `02-geocoding-parallel-with-r.R` file in the `Chapter09\R` folder.

If you run the code, you will get impressive results: 120 addresses were geocoded in *just 3 seconds* versus 30 seconds for sequential code! Also, in this case, you achieved a 10x improvement in running time thanks to `furrr`! Simple as that, right?

You can further simplify the code you just ran by adopting a geocoding package that does the bulk of the work for you in invoking the web service. Let's see what this is all about.

Using the tidygeocoder package in parallel

The `tidygeocoder` (<https://jessecambon.github.io/tidygeocoder/>) package provides a unified high-level interface for a selection of supported geocoder services and returns results in tibble format. Some of the providers available are Google, Mapbox, TomTom, and Bing (starting with version 1.0.3).

First of all, you need to install it. If you want to install the latest version, remember to reset the snapshot repository using the following code:

```
local({
  r <- getOption("repos")
  r["CRAN"] <- "https://cloud.r-project.org/"
  options(repos = r)
})
```

Then, you can simply run the `install.packages("tidygeocoder")` command in the Console window.

Thanks to Tidygeocoder, the `GET` request for an address is simply transformed into the `details_tbl <- geo(address, method = 'bing', full_results = TRUE)` command, and the great thing is that the returned object, `r`, already contains the attributes useful for geocoding, such as `details_tbl$bing_address.formattedAddress`. Therefore, the `bing_geocode_via_address()` function you encountered in the previous sections is embarrassingly simplified, as shown in the following code block:

```
bing_geocode_via_address <- function(address) {
  details_tbl <- geo(address, method = 'bing', full_results = TRUE)

  details_lst <- list(
    formattedAddress = details_tbl$bing_address.formattedAddress,
    lat = details_tbl$point.coordinates[[1]][1],
    lng = details_tbl$point.coordinates[[1]][2],
    details_tbl = details_tbl
  )
  return( details_lst )
}
```

Also, in this case, the `tidygeocoder` package selects the most appropriate match if the geocoding operation returns more than one, saving you from further headaches.

Important Note

Note that Tidygeocoder assumes that the `BINGMAPS_API_KEY` environment variable has been set and uses it to log in to the web service.

You can find the complete script that uses the Geocoder library to geocode addresses in parallel in the `03-geocoding-parallel-using-sdk-with-r.R` file in the `Chapter09\R` folder.

As you've probably figured out by now, using SDKs available to the community makes your life easier and is a winning choice. If an SDK for a specific web service is not available, you still learned how to make a raw `GET` request with R.

Let's now look at the benefits of what we've learned so far by implementing web service data enrichment solutions with R and Python in Power BI.

Accessing web services using Power BI

Power BI already has default features that allow you to access the data exposed by a web service into Power Query. There are two main modes:

- Via GUI (click on **Get data**, then **Web**, and then you can set advanced options if needed)
- Through the M language, using the `web.Contents()` function

The use of the GUI is very cumbersome and almost always it does not lead to the desired results. The only way to effectively connect to a web service using native Power BI features is to write M code. Writing code in M is not too difficult. However, there are some complications in using the `web.Contents()` function that arise when publishing a report that makes use of it to the Power BI service. In short, it is necessary to be careful when you have to build the URL to use in the `GET` request in a dynamic way, making use of the **relative path** and the **query options**. If you do not use this particular construct, the service will not be able to refresh the data. Moreover, the Power BI service does not allow you to securely store sensitive data, such as the API key, forcing you to embed this information into your code. In addition to that, multiple calls to an endpoint cannot be made in parallel using M.

It is for the reasons listed above that we suggest using R or Python to access web services, especially if SDKs are available to facilitate their use.

Keep in mind that the following restriction applies:

Important Note

If the report that uses web service data has to be published to the Power BI service, you can only query web services through Power Query and not within R visuals because the environment used by them on the service is not exposed on the internet.

That said, in order to be able to geolocate addresses using the Bing Maps Locations API in Power BI using the scripts we provide, the following clause applies:

Important Note

You must define the `BINGMAPS_API_KEY` environment variable as a user environment variable on your operating system to use the geocoding script we'll provide for Power BI. If the report that makes use of the data extracted from the web service will be published to the Power BI service, make sure to create the same environment variable on the data gateway machine as well.

Now, let's see how to extract data from a web service in Power BI with Python.

Geocoding addresses in Power BI with Python

In Power BI, we will use the Python code that calls the web service through the Geocoder SDK and exploits the parallelism thanks to Dask. The code to be used is practically identical to that already analyzed previously. Therefore, it is enough to follow these steps:

1. Open Power BI Desktop and make sure it references the `pbi_powerquery_env` environment.
2. Click on **Get data**, search and select **Python script**, and then copy the script you can find in the `04-geocoding-parallel-using-sdk-in-power-bi-with-python.py` file into the `Chapter09\Python` folder. Make sure to edit the path to the `geocoding_test_data.csv` file. Then click **OK**.
3. After a few seconds, you will see the **Navigator** window appear, where you can select the `enriched_df` table:



Figure 9.7 – The enriched DataFrame loaded in Power BI
Figure 9.7 – The enriched DataFrame loaded in Power BI

Then, click on **Load**.

You just queried the Bing Maps Locations API from within Power BI in parallel using Python! Easy, right?

It is possible to do the same thing with R. Let's see how to proceed.

Geocoding addresses in Power BI with R

In Power BI, we will use the R code that calls the web service through the Tidygeocoder SDK and that exploits the parallelism thanks to the `furrr` package. The code to be used is practically identical to that already analyzed previously. Therefore, it is enough to follow these steps:

1. Open Power BI Desktop and make sure it references your latest CRAN R.
2. Click on **Get data**, search and select **Python script**, and then copy the script you can find in the `04-geocoding-parallel-using-sdk-in-power-bi-with-r.R` file into the `Chapter09\R` folder. Make sure to edit the path to the `geocoding_test_data.csv` file. Then click **OK**.
3. After a few seconds, you will see the **Navigator** window appear, where you can select the `tbl_enriched` table:



Figure 9.8 – The tbl_enriched DataFrame loaded in Power BI
Figure 9.8 – The tbl_enriched DataFrame loaded in Power BI

Then, click on **Load**.

You just queried the Bing Maps Locations API from within Power BI in parallel using R too!

Summary

In this chapter, you learned how to query RESTful web services using Python and R. In addition to learning how to execute raw `GET` requests with both languages, you've also learned how to parallelize multiple calls to the same endpoint by taking advantage of the multithreading capabilities of your machine. Moreover, you've also come across some SDKs of the Bing Maps Locations API, both for Python and R, which make accessing the data much easier. Finally, you've seen how all of this is easily implemented in Power BI.

In the next chapter, you'll see how to enrich your data by applying complex algorithms to the data you already have. In this way, you will create new variables that give a new light to your data, making it more useful to reach your goal.

References

For additional reading, check out the following books and articles:

1. *What RESTful actually means* (<https://codewords.recurse.com/issues/five/what-restful-actually-means>)
2. *Bing Maps REST Services* (<https://docs.microsoft.com/en-us/bingmaps/rest-services>)
3. *A Future for R: A Comprehensive Overview* (<https://cran.r-project.org/web/packages/future/vignettes/future-1-overview.html>)
4. [VIDEO] *Accessing API and web service data using Power Query* (<https://www.youtube.com/watch?v=SoJ52o7ni2A>)
5. *Invoking M Functions In Parallel Using List.ParallelInvoke()* (<https://blog.crossjoin.co.uk/2018/09/20/invoking-m-functions-in-parallel-using-list-parallelinvoke/>)

10 Calculating Columns Using Complex Algorithms

The data ingestion phase allows you to gather all the information you need for your analysis from any data source. Once the various datasets have been imported, it may be that some of this information, taken as it is, isn't useful in describing a phenomenon from an analytical point of view. It is often necessary to apply non-trivial algorithms to the data you have in order to get measures or indicators that will do the trick and Power BI often doesn't have the tools to calculate them. Fortunately, thanks to R and Python, we have everything we need to calculate our measures.

In this chapter, you will learn about the following topics:

- The distance between two geographic locations
- Implementing distances using Python
- Implementing distances using R
- The basics of linear programming
- Definition of the LP problem to solve
- Handling optimization problems with Python
- Solving LP problems with R

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

The distance between two geographic locations

It often happens that you have in your dataset coordinates expressed in longitude and latitude that identify points on the globe. Depending on the purpose of the analysis you need to complete, you can leverage these coordinates to calculate measures that best help to describe the scenario you want to deal with. For example, assuming you have the geographic coordinates of some hotels in a dataset, it might make sense to calculate the distance of each of them to the nearest airport if you want to give an additional value of interest to a visitor. Let's start by figuring out what types of distances to consider for our case.

Spherical trigonometry

The study of how to measure triangles (**trigonometry**) has been of great interest in the past. The ancient Egyptians and Babylonians had already addressed the issues between the relationships between sides, although they did not yet have the notion of an angle. It is thanks to **Hellenistic mathematics** that the concepts of trigonometric functions as we know them now began to spread around the world, even reaching India and China.

It was the ancient Greeks who, once they had explored all the properties associated with a triangle drawn on a plane, came up with the idea of imagining a triangle drawn on a sphere. The importance of measuring distances between points on a sphere was immediately shown to be of interest in later centuries for navigation and astronomy. Therefore, several minds devoted themselves eagerly to the discovery of important properties that today can be collected under the name of **spherical trigonometry**.

If you draw a spherical triangle, you will immediately notice the differences from a flat triangle:

Figure 10.1 – A spherical triangle
Figure 10.1 – A spherical triangle

The main difference with planar triangles is that the sides of spherical triangles are arcs of **great circles** or **geodesics** (circumferences that always divide the sphere in half), and since the central angles (near the center, O) are proportional to the lengths of their respective arcs ($\text{length} = \pi \times \text{angle}$), the sides a , b , and c are measured with *angle units* rather than linear units. If you want a visualization of the fact that the sides of the spherical triangle belong to three great circles, *Figure 10.2* can help you:



Figure 10.2 – The great circles that generate a spherical triangle
Figure 10.2 – The great circles that generate a spherical triangle

The mathematics describing spherical trigonometry makes it possible to define all the distances between two points on the sphere so far highlighted by great mathematicians of the past.

As for the distances between two geographical points, the reference coordinate system will be the one that makes use of **latitude** and **longitude**. Obviously, we will not go into the mathematical detail of the proofs of distances that we will propose shortly (also because some of them would be very complex indeed). We did, however, want to provide an introduction that would lay the groundwork that can be found in the most frequently used concepts of geographic distances.

Let's now explore the most straightforward distance between two points, the law of Cosines distance.

The law of Cosines distance

The **law of Cosines distance** (also called the **great circle distance**) is the shortest distance between two points on the surface of a sphere, measured along the surface of the sphere. Given two points, P and Q , a unique great circle passes through them. The two points separate the great circle into two distinct arcs. The length of the shorter arc is the distance of the great circle between the points:



Figure 10.3 – The great circle distance between the points P and Q
Figure 10.3 – The great circle distance between the points P and Q

Observe that if the two points of which to calculate the distance are at the antipodes of the sphere (the **antipodal points** u and v), then the great circles that pass through them are infinite and the distance between these two points is calculated very easily, as it measures exactly half the circumference of the sphere.

When, on the other hand, the points are not antipodal, it is possible to derive the preceding distance thanks to the **spherical law of Cosines** (see *References* at the end of this chapter) that governs spherical trigonometry (that's why we also call it the law of Cosines distance). Without tediously going through the mathematical steps, the formula that calculates the law of Cosines (or great circle) distance between two points on a sphere is as follows:



Figure 10.4 – The formula of the law of Cosines distance between two points
Figure 10.4 – The formula of the law of Cosines distance between two points

Looking at it, it's not the simplest, cleanest formula you've ever seen, right? Despite this, having a handy calculator available today, the preceding calculation remains feasible. Now imagine the ancient navigators who used to apply the great circle distance between various points on the globe. How could they have used the preceding formula, even though they had sine and cosine tables that would have facilitated some calculations? It would have been a very complex activity and subject to errors that could have cost the sailors their lives.

Important Note

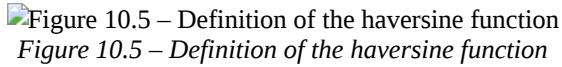
If the two points are close together (for example, a few kilometers apart on the sphere) and you don't have a calculator with accurate precision, you might get inaccurate results.

For these reasons, mathematicians of the age introduced the new trigonometric `haversin` function (or `hav`), which allows you to transform and smooth out the great circle distance formula, also avoiding the previously

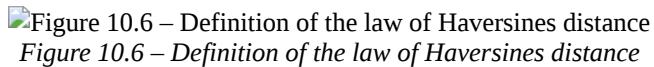
mentioned error for small distances. Let's see how it does this.

The law of Haversines distance

The function called `haversine` (from *half-versed sine*) is defined as follows:



Thanks to this new function, it is possible to rewrite the law of Cosines distance as follows:



This new distance formulation is known as the **law of Haversines distance**. Its undoubted usefulness to navigators of the time becomes clear when you consider that, along with the tables of sines and cosines, tables of Haversines were also published. Therefore, the calculation of the distance between two points became immediate.

Important Note

Even this formula suffers from rounding errors due to the special (and somewhat unusual) case of antipodal points.

To get a distance formula that has better accuracy than the Haversines one, you have to use Vincenty's formula. Let's see what this is all about.

Vincenty's distance

The winning assumption that led the geodesist Vincenty to a more precise formula of the distance between two points was to consider the Earth as not a sphere but an ellipsoid slightly flattened at the poles (the difference is only about 21 km):



Unlike the Haversines method for calculating a distance on a sphere, **Vincenty's formulas** describe a method that needs to converge to a solution through several iterations. In detail, a sequence of equations is calculated whose output is fed back into the same sequence of equations with the goal of minimizing the calculated value after a certain number of iterations. For this reason, Vincenty's formulas are computationally more demanding.

Vincenty's formulas are related to two problems:

- **Direct problem:** Find the endpoint, (Φ_2, L_2) , and azimuth, α_2 , given an initial point, (Φ_1, L_1) , and initial azimuth, α_1 , and a distance, s .
- **Inverse problem:** Find the azimuths α_1 , α_2 and the ellipsoidal distance, s , given the coordinates of the two points, (Φ_1, L_1) and (Φ_2, L_2) .

The reverse problem is what we are interested in as we need to calculate the distance. To get an idea of the complexity of the formulas, take a look at the *References* section at the end of the chapter.

Vincenty's formulas are widely used in projects where there is a need for high precision in measurements because they are accurate to within 0.5 mm (0.020 in) of the Earth's ellipsoid.

Important Note

If the points are nearly antipodal, the algorithm fails to converge and the error is much larger or converges slowly.

In 2013, Karney used Newton's method to give rapid convergence for all pairs of input points without any error.

At this point, the question that arises is what kind of distance is best to use and when. Let's try to understand that in the following section.

What kind of distance to use and when

Considering the strong limitation that the law of Cosines distance has for short distances between two points, the most used methods today in common applications are the law of Haversines (for short, Haversine) and Vincenty's formulas (for short, Vincenty). Here are our suggestions about which distance to use in particular scenarios:

- For *points located close to each other* (think short-range flights), the approximation of the Earth to a sphere is very likely. Therefore, methods based on the spherical Earth model, such as Haversine, which are computationally simpler (hence faster), will be quite adequate.
- For *points located far away* (such as long-range flights, especially connecting opposite hemispheres), the spherical Earth model starts to be less tight and inaccurate. In these cases, Vincenty's inverse formula for ellipsoids, which is substantially more computationally complex (hence generally slower), will give a better result. If you have computational limitations, you need to consider whether the faster models give sufficiently accurate results for your purposes.

Now that you've become well versed in the theory, let's move on to implementing these algorithms in Python and R.

Implementing distances using Python

The scenario on which we will implement the distance algorithms just described involves a dataset of US hotels, containing the latitude and longitude of each. The goal is to enrich the dataset by adding the distances to the nearest airports.

The hotel data is publicly available on *Back4App* (<https://bit.ly/data-hotels-usa>). For convenience, we extracted only 100 hotels from New York City and we will calculate for each of them the distances from the LaGuardia and John F. Kennedy airports (you can find the airport data here: <https://datahub.io/core/airport-codes>) using the Haversine (spherical model) and Karney (ellipsoidal model) methods. You can find the already extracted datasets for your convenience in the `Chapter10` folder of the GitHub repository. In detail, you will find the hotel data in the `hotels-ny.xlsx` file and the airport data in the `airport-codes.csv` file.

Calculating distances with Python

As we mentioned earlier, we are not those that like to reinvent the wheel, especially when there is a risk of running into complexities related to the domain of the problem to be solved. Fortunately, Python has a very active community of programmers with expertise in specific scientific domains who share their artifacts publicly. This is the case of the `PyGeodesy` package (<https://github.com/mrJean1/PyGeodesy>), created by Jean M. Brouwers, which implements in pure Python various computational tools for spherical and ellipsoidal models of the Earth that Chris Veness has made available for Java and Charles Karney himself has made available in C++.

To be able to use this module, you must obviously install it in your environment and, since we intend to use the distance formulas optimized by Karney, we must also install the `geographiclib` package, directly maintained by him. Therefore, proceed as follows:

1. Open Anaconda Prompt.
2. Switch to the `pbi_powerquery_env` environment, entering the `conda activate pbi_powerquery_env` command, and then press *Enter*.
3. Install the `PyGeodesy` package, entering `pip install PyGeodesy`, and then press *Enter*.

4. Install the `geographiclib` package, entering `pip install geographiclib`, and then press *Enter*.
5. If you have not already done so, also install the `openpyxl` package by entering `pip install openpyxl`, and then press *Enter*.

At this point, you can proceed with the Python code. First, note that `PyGeodesy` package contains a form with basic geodesic functions, called `formy`. In this module, there are functions that directly calculate distances according to the Haversine's and Vincenty formulas, but it doesn't contain the variant of Karney's formulas. Therefore, in addition to the standard `pandas` and `NumPy` modules, the following must be imported:

```
from pygeodesy import formy as frm
from pygeodesy.ellipsoidalKarney import LatLon as kLatLon
```

To calculate the distance according to Karney, you must use the objects provided by the `ellipsoidalKarney` module. Basically, you have to create the two points on the ellipsoid using the `LatLon` method of this model, and then calculate the distance. This is summarized in the following `karney` user-defined function:

```
def karney(lat1, lng1, lat2, lng2):
    return kLatLon(lat1, lng1).distanceTo(kLatLon(lat2, lng2))
```

After that, a second user-defined function is created for convenience as a wrapper for the calls to the calculation of the various distances:

```
def geodistance(lat1, lng1, lat2, lng2, func):
    return func(lat1, lng1, lat2, lng2)
```

Then, the hotels data is imported into the `hotel_df` dataframe and the airports data into the `airports_df` one. Since the airports dataframe has the `coordinates` column, which contains a string with longitude and latitude separated by a comma, these two values are split into two separate columns using the `split()` function and then appended to the same source dataframe without the `coordinates` column, now useless:

```
airports_df = pd.concat([
    airports_df.drop(['coordinates'], axis=1),
    airports_df['coordinates'].str.split(',', expand=True).rename(columns={0:'longitude', 1:'latitude'})])
```

In order to conveniently access the latitude and longitude values of a specific airport, the user-defined `airportLatLongList()` function has been created, which accepts as parameters both a dataframe containing the airport data with the `iata_code`, `latitude` and `longitude` columns and the specific **IATA code** of the airport of interest. Remember that the IATA airport code is a three-letter code that identifies many airports and metropolitan areas around the world, defined by the **International Air Transport Association (IATA)**. Therefore, John F. Kennedy International Airport is identified by the IATA code `JFK` and LaGuardia Airport by the code `LGA`. So, in order to get the coordinates of those airports, you can use the following code:

```
jfk_lat, jfk_long = airportLatLongList(airports_df, 'JFK')
lga_lat, lga_long = airportLatLongList(airports_df, 'LGA')
```

That said, thanks to the `geodistance()` function, it is enough to have the geographical coordinates of two points in order to calculate the distance between them. For example, if you want to calculate the haversine distance between point A(`lat1,lng1`) and point B(`lat2,lng2`), you just have to use this code:

```
geodistance(lat1, lng1, lat2, lng2, func=frm.harvesine)
```

To be able to calculate the Karney distance between them instead, you can take advantage of the `karney()` function and use this code:

```
geodistance(lat1, lng1, lat2, lng2, func=karney)
```

However, how should you proceed if you no longer want to apply the `geodistance()` function to two single points, but to a series of points contained in a dataframe column and a second fixed point? Since the preceding function needs five input parameters, we could have used the `apply()` method of the `pandas` dataframe (as shown here: <http://bit.ly/pandas-apply-lambda>). Instead, we introduced a convenient way to evaluate a function over successive tuples of the input Series. In order to vectorize a function, you must invoke the `np.vectorize()`

method and pass as a parameter the function to be applied to the geographic coordinate Series. Then, you also have to pass the parameters of the input function as follows:

```
hotels_df['haversineDistanceFromJFK'] = np.vectorize(geodistance)(  
    hotels_df['latitude'],  
    hotels_df['longitude'],  
    jfk_lat,  
    jfk_long,  
    func=frm.haversine)
```

The distances (in meters) resulting from the previous calculation are stored into the new `haversineDistanceFromJFK` column of the `hotels_df` datafram. Similarly, the Karney distance can be calculated by simply referencing the `karney` function in the code chunk.

Important Note

A **vectorized function** is not the same as a function used with `np.vectorize()`. A vectorized function is a function built into NumPy and executed in the underlying compiled code (C or Fortran) so that special processor registers are used to operate on several items at once. As you can imagine, vectorization is much more performant and preferable to `for` loops. For more details, check out the *References* section.

If you run the code for the `01-distances-from-airports-in-python.py` file in the `Python` folder, you'll get something like this:

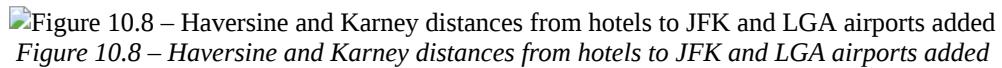


Figure 10.8 – Haversine and Karney distances from hotels to JFK and LGA airports added
Figure 10.8 – Haversine and Karney distances from hotels to JFK and LGA airports added

Amazing! You were able to calculate both the Haversine and Karney distances in meters between all hotels and both airports using Python. At this point, it is straightforward to use similar code to calculate distances in Power BI. Let's see how to do it.

Calculating distances in Power BI with Python

It's time to implement what you've learned in Power BI. So, launch Power BI Desktop and let's get going:

1. Make sure Power BI Desktop is referencing your latest environment in **Options**. After that, click on **Excel** to import the `hotels-ny.xlsx` file, which you can find in the `Chapter10` folder. Select it and click **Open**.
2. Select the **Sheet 1** table from the **Navigator** window:

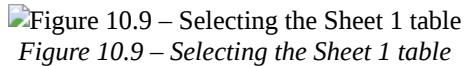
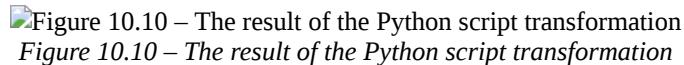


Figure 10.9 – Selecting the Sheet 1 table
Figure 10.9 – Selecting the Sheet 1 table

Then, click on **Transform data**.

3. Click on the **Transform** menu and then click on **Run Python Script**.
4. Copy the script from the `02-distances-from-airports-in-power-bi-with-python.py` file from the `Chapter10\Python` folder into the Python script editor and click **OK**.
5. You may be prompted to align the permissions in the Excel file with those you initially selected for the scripts (in our case, **Organizational**). In this case, you already know how to proceed based on what you've seen in *Chapter 5, Using Regular Expressions in Power BI*.
6. We are only interested in the data in `dataset`. So, click on its **Table** value.
7. Power Query will transform your data by adding the distances from each hotel to the two airports, `JFK` and `LGA`, for the two methodologies of Haversines and Karney:



8. You can then click **Close & Apply** in the **Home** tab.

Great! You just enriched your data by adding the distances between two geographical points in Power BI using Python. Let's see how to do that in R.

Implementing distances using R

The scenario will be the same as the one already described in the previous section. We will therefore enrich the data relating to some hotels in New York City with the distances separating them from the two major airports of New York, namely John F. Kennedy and LaGuardia.

The files containing the data to be processed can be found in the `Chapter10` folder of the GitHub repository. In detail, you will find the hotels data in the `hotels-ny.xlsx` file and the airports data in the `airport-codes.csv` file.

Calculating distances with R

The R community is also fortunate to have a freely available package that implements spherical trigonometry functions for geographic applications. The package is called `geosphere` (<https://cran.r-project.org/web/packages/geosphere/>) and, like the Python `PyGeodesy` package, it is inspired by the code that Chris Veness and Charles Karney have made publicly available.

First, you need to install this new package:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('geosphere')`. Then, press *Enter*.

You are now ready to develop your code in R. Apart from the usual packages that allow you to read CSV and Excel files (`readr` and `readxl`) and facilitate data transformation operations (`dplyr` and `purrr`), you must, of course, load the package you just installed.

You can easily import hotel data into the `hotels_tbl` tibble using the `read_xlsx()` function and airport data into the `airport_tbl` tibble using the `read_csv()` function. At this point, the first operation to do is to split the contents of the `coordinates` column of `airports_tbl` in the two new columns, `longitude` and `latitude`:

```
airports_tbl <- airports_tbl %>%  
tidy::separate(  
col = coordinates,  
into = c('longitude', 'latitude'),  
sep = ',',  
remove = TRUE,  
convert = TRUE )
```

Note the simplicity of using the `separate` function from the `tidyverse` package:

1. The pipe passes the `airports_tbl` tibble as the first parameter for the function.
2. Declare the column to be split (`col = coordinates`).
3. Declare the two new target columns (`into = c('longitude', 'latitude')`).
4. Declare the separator found in the values of the column to be split (`sep = ',', '`).
5. Remove the column to be split when the transformation is complete (`remove = TRUE`).
6. Let the data type of the target columns convert automatically if they are numeric columns (`convert = TRUE`).

All this in one operation with maximum clarity. That's one of the reasons data analysts love R!

Again, we make use of a function to conveniently access the longitude and latitude values of a specific airport. It is the `airportLongLatVec()` function and it accepts as parameters both a dataframe containing the airport data with the `iata_code`, `latitude`, and `longitude` columns, and the specific **IATA code** of the airport of interest:

```
airportLongLatVec <- function(df, iata) {  
  ret_vec <- df %>%  
    filter( iata_code == iata ) %>%  
    select( longitude, latitude ) %>%  
    unlist()  
  return(ret_vec)  
}
```

The output is a named vector. So, the coordinates of the two airports can be easily found in this way:

```
jfk_coordinates <- airportLongLatVec(airports_tbl, 'JFK')  
lga_coordinates <- airportLongLatVec(airports_tbl, 'LGA')
```

You are pretty much ready to transform the data. From the `geosphere` package, you will use the `distHaversine()` and `distGeo()` functions. The former is self-explanatory from the name itself. The `distGeo()` function calculates the shortest distance between two points on an ellipsoid according to Karney's formulas. Both functions accept two pairs of coordinates (in the order longitude and latitude) in vector form. To get the same results as Python, the `distHaversine()` function must accept as a parameter the same mean radius of the Earth sphere model used by default by PyGeodesy. The radius in question is **R1** (mean radius) defined by the **International Union of Geodesy and Geophysics (IUGG)**, which is 6,371,008.771415 meters.

At this point, the enrichment operation of the `hotels_tbl` tibble can be done using the already seen family of `map()` functions of the `purrr` package. In the first step, we create a new column, `p1`, containing the longitude and latitude pairs in a vector using the `map2()` function. In the second step, we apply the `distHaversine()` and `distGeo()` functions to the newly created point, `p1`, and to the fixed points identifying the airports (`jfk_coordinates` and `lga_coordinates`) to create the new columns containing the distances. This is the code needed:

```
hotels_tbl <- hotels_tbl %>%  
  mutate(  
    p1 = map2(longitude, latitude, ~ c(.x, .y))  
  ) %>%  
  mutate(  
    haversineDistanceFromJFK = map_dbl(p1, ~ distHaversine(p1 = .x, p2 = jfk_coordinates, r = 6371008),  
    karneyDistanceFromJFK = map_dbl(p1, ~ distGeo(p1 = .x, p2 = jfk_coordinates)),  
    haversineDistanceFromLGA = map_dbl(p1, ~ distHaversine(p1 = .x, p2 = lga_coordinates, r = 6371008),  
    karneyDistanceFromLGA = map_dbl(p1, ~ distGeo(p1 = .x, p2 = lga_coordinates))  
  ) %>%  
  select( -p1 )
```

Recall that the `map2()` function takes two vectors as input and runs them in parallel to pass their values to the function used after the `~` symbol (in our case, the `c()` function that declares a vector). The `map_dbl()` function instead takes as input the column `p1` (which contains the geographic coordinates in vector format of the hotels) and passes its elements to the function after the `~` (in our case, `distGeo()` with other fixed parameters) transforming the output into a vector of double numeric data types.

If you run the code for the `01-distances-from-airports-in-r.R` file in the `R` folder, you'll get something like this:

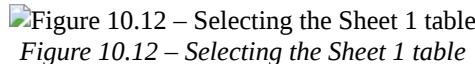
Figure 10.11 – The enriched hotels tibble with distances
Figure 10.11 – The enriched hotels tibble with distances

Wow! You were able to calculate both the Haversine and Karney distances between all hotels and both airports also using R. At this point, it is straightforward to use similar code to calculate distances in Power BI. Let's see how to do it.

Calculating distances in Power BI with R

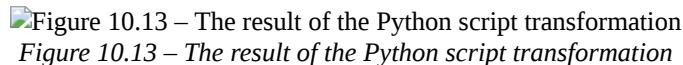
It's time to implement what you've just learned in Power BI. So, launch Power BI Desktop and let's get going:

1. Make sure Power BI Desktop is referencing your latest environment in **Options**. After that, click on **Excel** to import the `hotels-ny.xlsx` file, which you can find in the `Chapter10` folder. Select it and click **Open**.
2. Select the **Sheet 1** table from the **Navigator** window:

Figure 10.12 – Selecting the Sheet 1 table
Figure 10.12 – Selecting the Sheet 1 table

Then, click on **Transform data**.

3. Click on the **Transform** menu and then click on **Run R Script**.
4. Copy the script from the `02-distances-from-airports-in-power-bi-with-r.R` file from the `Chapter10\R` folder into the R script editor and click **OK**.
5. We are only interested in the data in `hotels_df`. So, click on its **Table** value.
6. Power Query will transform your data by adding the distances from each hotel to the two airports, `JFK` and `LGA`, for the two methodologies of Haversines and Karney:

Figure 10.13 – The result of the Python script transformation
Figure 10.13 – The result of the Python script transformation

7. You can then click **Close & Apply** in the **Home** tab.

Great! You just enriched your data by adding distances between two geographical points in Power BI using R.

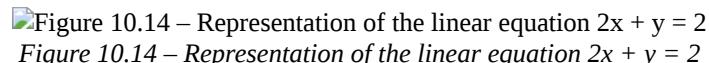
In the next section, you will see how to enrich your dataset using linear optimization algorithms.

The basics of linear programming

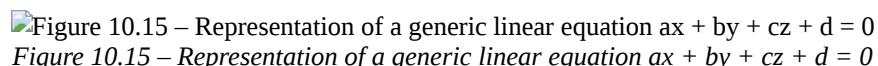
Linear Programming (LP) algorithms are adopted in all those areas where optimization, and therefore the economy of resources, is critical to the continuation of activities. In order to understand what this is all about, you need some math. In this way, let's brush up on some of the geometric concepts we encountered during our youthful studies.

Linear equations and inequalities

We all at least once in our lives have encountered the term linear equation. A **linear equation**, in its simplest sense, consists of a mathematical relationship between two variables, x and y , in the form $ax + by + c = 0$, which on the Cartesian plane identifies a **straight line**:

Figure 10.14 – Representation of the linear equation $2x + y = 2$
Figure 10.14 – Representation of the linear equation $2x + y = 2$

Evidently, the variables involved in a linear equation can be more than two. The representation of a linear equation is possible as long as we have three variables (the famous three dimensions we can see). In this case, a linear equation of three variables, in the form $ax + by + cz + d = 0$, represents a **plane** in the space:

Figure 10.15 – Representation of a generic linear equation $ax + by + cz + d = 0$
Figure 10.15 – Representation of a generic linear equation $ax + by + cz + d = 0$

When you have more than three variables in a linear equation, we commonly call its representation no longer a plane, but a **hyperplane**.

There are also **linear inequalities**, which are linear functions involving inequalities (identified by the symbols $<$, $>$, \leq , and \geq). Just as done for linear equations, in the same way, you can plot linear inequalities, either with two variables or with three. They represent all points on either side of the line in the case of two variables (that is, a region of the plane), or points on either side of the straight plane in the case of three variables (that is, a volume):

Figure 10.16 – Representation of generic linear inequalities with two or three variables
Figure 10.16 – Representation of generic linear inequalities with two or three variables

Observe that the regions identified by the inequalities are infinite but have an edge constituted by the linear equation that derives from the same inequality considering the $=$ sign instead of the inequality sign.

Well, very often when you think about these concepts, you just associate them with theoretical mathematical stuff, but that's not the case. Simple concepts related to transportation, manufacturing, shipping, and so on can be traced back to linear equations or inequalities. Let's see an example.

Formulating a linear optimization problem

Imagine you work in a manufacturing company and you need to produce two products, P1 and P2. To produce them, you need a machine. Specifically, producing one unit of product P1 takes 30 minutes of processing time on the machine, while producing one unit of product P2 takes 25 minutes on the same machine. In addition, the number of hours for which the machine, M, can remain on is 40 hours ($= 40 \times 60$ minutes). That is, 30 minutes multiplied by the number of P1 products added to 25 minutes multiplied by the number of P2 products cannot exceed 40 hours of processing time. So, given x number of products P1 and y number of products P2 produced at the end of processing, the machine hours **constraint** can be summarized as follows:

Wow! You just used a simple linear inequality to describe a constraint on a manufacturing process.

Now imagine that you have received demands for P1 and P2 products from customers. Specifically, adding up all the requests, you need to produce at least 45 units of P1 and 25 units of P2 to satisfy your customers. These demand constraints can be summarized as follows:

Awesome! You've added two more linear constraints to your problem. If you add a goal to these business constraints, for example, you want to maximize the total number, z , of units of P1 and P2, the set of constraints and goal constitute a linear optimization problem:

Simple as that, right? If we want to be a little more formal, LP (also called **linear optimization**) consists of a set of techniques useful for solving systems of linear equations and inequalities with the goal of maximizing or minimizing a linear objective function. In particular, the variables x and y are called **decision variables** and the objective that is set is called the **objective function** or **cost function**. In this example, the business case required only inequality constraints, but there can also be equality constraints.

Cool! But now that we've set up the problem from a mathematical standpoint, how do we solve it? First, we need to take all the inequalities and represent them on the axes (in this case you can, because they only contain two variables). As seen at the beginning, a linear inequality represents a portion of a plane bounded by the straight line considering the sign of equality in the inequality itself. Intersecting all these planes, we come to identify an area common to all the inequalities, called the **feasible region**. Basically, all points that are in this region satisfy all the constraints identified by the inequalities.

If you want to draw the feasible region associated with the constraints of the example we have just illustrated, there is no need to go over all the geometry studied in secondary school, but just use **WolframAlpha** (<https://www.wolframalpha.com/>). Enter the following string in the search engine:

plot $30x+25y \leq 2400$ and $x \geq 45$ and $y \geq 25$. All constraints must be satisfied simultaneously, hence the use of the and operator. Press *Enter* and you will see this result:

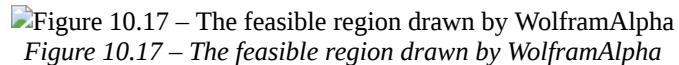


Figure 10.17 – The feasible region drawn by WolframAlpha
Figure 10.17 – The feasible region drawn by WolframAlpha

To the WolframAlpha result, we have added the values of the vertices of the feasible region, which is a triangle.

Important Note

It is shown that if the optimization problem is solvable, the solution that maximizes or minimizes the objective function lies precisely at one of the vertices of the feasible region.

As you can see in *Figure 10.17*, the x value of vertex C is 59.1. Clearly, it is not possible to have a fractional part of the product P1 since it is not possible to produce only a fraction of one unit. The nearest integer value should then be considered, which is 59. If, therefore, the business problem imposes to have **integer values** for the decision variables, then the problem becomes one of **Mixed-Integer Linear Programming (MILP)**. In our case, then, we consider vertices A, B, and $C^* = (59, 25)$ as possible solutions to our problem and substitute their coordinates into the objective function, $z = x + y$:

- A: $z = 45 + 25 = 70$
- B: $z = 45 + 42 = 87$
- C^* : $z = 59 + 25 = 84$

We infer that the solution to our problem is given by the vertex B with a maximum value of 87 units, namely $P1 = 45$ and $P2 = 42$.

Note

In summary, producing 45 units of product P1 and 42 units of product P2 satisfies the machine's hourly production constraint and the customer demand constraints, while maximizing total production.

Did you ever think you could solve a linear optimization problem before today? Well, you've done it! Clearly, it is possible to solve these problems by hand when they are this simple. But when the number of decision variables grows, it is no longer possible to draw the feasible region and therefore it becomes impossible to locate the vertices in a multidimensional space by eye. In these cases, it is mathematics and, above all, the packages made available by the community for Python and R that allow us to find the solution to the problem. Let's first look at a slightly more complex case of an LP problem.

Definition of the LP problem to solve

Now imagine that you are working for a company that needs to ship a product from different warehouses around the world to different countries. You have to hand the following:

- The quantities of product available in warehouses:

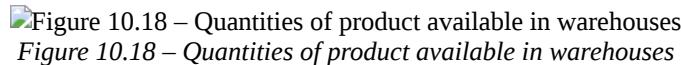


Figure 10.18 – Quantities of product available in warehouses
Figure 10.18 – Quantities of product available in warehouses

- The quantities of product required by countries:

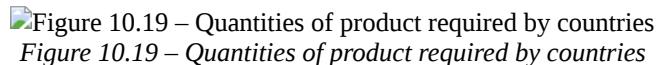


Figure 10.19 – Quantities of product required by countries
Figure 10.19 – Quantities of product required by countries

- The shipping costs from each of the warehouses to all requesting countries:



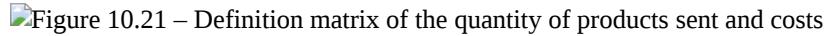
Figure 10.20 – Costs from warehouses to countries
Figure 10.20 – Costs from warehouses to countries

Figure 10.20 – Costs from warehouses to countries

Your goal is to minimize your company's costs by meeting all customer demands from different countries.

Formulating the LP problem

As seen in the previous section, you must first formulate the problem mathematically. Let's use a couple of numerical indexes, i and j , to identify the quantities sent and costs. In detail, consider the quantity x_{ij} of product that is shipped from *Warehouse i* to *Country j* according to this matrix that defines the decision variables and costs:


Figure 10.21 – Definition matrix of the quantity of products sent and costs

The quantity x_{ij} is an integer and non-negative () .

Given the preceding definitions, the target of the problem is to minimize the objective function, which can be written as the sum of the product between the cost of shipping from *Warehouse i* to *Country j* (C_{ij}) and the quantity of product shipped from *Warehouse i* to *Country j* (x_{ij}):

Written in full and organizing it by warehouses for reading convenience, taking the cost amounts from *Figure 10.21*, the previous objective function can be rewritten as follows:

To this point, you must formalize the constraints, which are of two types: the *warehouse supply constraints* and the *customer demand constraints*:

- **Warehouse supply constraints:** Once a warehouse has been fixed (for example, the *Warehouse ITA*, for which $i = 1$), the sum of the products shipped from that warehouse to all countries (sum of x_{1j}) cannot exceed the maximum quantity of products contained in that warehouse (for *Warehouse ITA*, 50,000 products; see *Figure 10.18*). That is, for all six countries, we have the following: You will, therefore, have a similar constraint for each warehouse (four constraints in all).
- **Customer demand constraints:** Regardless of which warehouse the goods come from, you have to meet the demand for each country's products. Therefore, the sum of the products sent from all the warehouses toward a given country (for example, *France*, for which $j = 2$) must be at least equal to the demand of that country (France demands at least 15,000 products; see *Figure 10.19*). And, therefore, considering all four warehouses, we have the following: You will, therefore, have a similar constraint for each country (six constraints in all).

Therefore, the final linear optimization problem can be formulated as follows:

Awesome! You managed to formulate a non-trivial business problem in mathematical terms. Let's now see how to solve it with Python.

Handling optimization problems with Python

As you've probably already figured out, the large community that develops Python packages never stands still. Even in this case, it provided a module that helps us solve linear optimization problems. Its name is **PuLP** (<https://github.com/coin-or/pulp>) and it is an LP modeler written in Python. It interfaces with the most common free and not-free engines that solve LP, **Mixed Integer Programming (MIP)**, and other related problems, such as **GNU Linear Programming Kit (GLPK)**, **Coin-or Branch and Cut (CBC)**, which is the default one, and **IBM ILOG CPLEX**. Its use is quite straightforward. Let's put it into practice right away with the problem from the previous section.

Solving the LP problem in Python

The code that will be explained to you in this section can be found in the `03-linear-optimizaiont-in-python.py` file in the `Chapter10\Python` folder of the repository.

First, you have to install the `PuLP` module in your environment:

1. Open Anaconda Prompt.
2. Enter the `conda activate pbi_powerquery_env` command.
3. Enter the `pip install pulp` command.

After that, you can then define the values that will make up the constraints and costs using NumPy vectors and matrices:

```
import pandas as pd
import numpy as np
import pulp as plp
warehouse_supply_df = pd.read_excel(r'D:\<your-path>\Chapter10\RetailData.xlsx', sheet_name='Warehouse Supply')
warehouse_supply = warehouse_supply_df['product_qty'].to_numpy()
country_demands_df = pd.read_excel(r'D:\<your-path>\Chapter10\RetailData.xlsx', sheet_name='Country Demands')
country_demands = country_demands_df['product_qty'].to_numpy()
cost_matrix_df = pd.read_excel(r'D:\<your-path>\Chapter10\RetailData.xlsx', sheet_name='Shipping Costs')
n_warehouses = cost_matrix_df.nunique()['warehouse_name']
n_countries = cost_matrix_df.nunique()['country_name']
cost_matrix = cost_matrix_df['shipping_cost'].to_numpy().reshape(n_warehouses, n_countries)
```

In the script file, you will also find the code to import the values directly from the `RetailData.xlsx` file in the `Chapter10` folder.

It is possible then to define an `LpProblem` object, giving it a name and the type of optimization you want to apply to the objective function (minimize or maximize):

```
model = plp.LpProblem("supply-demand-minimize-costs-problem", plp.LpMinimize)
```

To this empty object, you can add an objective function and constraints later.

In order to construct the objective function, we must first define the decision variables (x_{ij}) via the `LpVariable` function, which accepts the variable name, the full list of strings representing the variable indices, the category of the variable (continuous, integer, or binary), and any upper- or lower-bound values. The index list is simply constructed with a nested **list comprehension** (<http://bit.ly/nested-list-comprehensions>):

```
var_indexes = [str(i)+str(j) for i in range(1, n_warehouses+1) for j in range(1, n_countries+1)]
print("Variable indexes:", var_indexes)
```

This is the output as an example:

```
Variable Indices: ['11', '12', '13', '14', '15', '16', '21', '22', '23', '24', '25', '26', '31', '32', '33', '34', '35', '36']
```

It is now possible to easily define the decision variables as follows:

```
decision_vars = plp.LpVariable.matrix(
    name="x",
    indexs=var_indexes,
    cat="Integer",
    lowBound=0 )
```

Since the decision variables are to be multiplied by the C_{ij} costs defined earlier in `cost_matrix`, it is appropriate to format the `decision_vars` list in the same shape as the cost matrix in order to be able to perform element-wise multiplication, also known as the **Hadamard product**:

```
shipping_mtx = np.array(decision_vars).reshape(n_warehouses, n_countries)
print("Shipping quantities matrix:")
print(shipping_mtx)
```

It returns the following output:

```
Shipping quantities matrix:
[[x_11 x_12 x_13 x_14 x_15 x_16]
```

```
[x_21 x_22 x_23 x_24 x_25 x_26]
[x_31 x_32 x_33 x_34 x_35 x_36]
[x_41 x_42 x_43 x_44 x_45 x_46]]
```

The objective function is then defined as the sum of the element-wise product of cost and shipping matrices:

```
objective_func = plp.lpSum(cost_matrix * shipping_mtx)
print(objective_func)
```

The output is the following one:

```
8*x_11 + 18*x_12 + 14*x_13 + 40*x_14 + 40*x_15 + 25*x_16 + 12*x_21 + 10*x_22 + 8*x_23 + 18*x_24 -
```

If you remember correctly, this expression coincides with the objective function written in full that you saw in the previous section.

You can then add the objective function to the model, as follows:

```
model += objective_func
```

Constraint inequalities are also added in the same way:

```
for i in range(n_warehouses):
    model += plp.lpSum(shipping_mtx[i][j] for j in range(n_countries)) <= warehouse_supply[i], "WareHouse"
for j in range(n_countries):
    model += plp.lpSum(shipping_mtx[i][j] for i in range(n_warehouses)) >= country_demands[j], "Country"
```

Finally, we can move on to solving the problem by running this simple script:

```
model.solve()
```

The first thing to do is to check the state of the solution, which can take the values `Optimal`, `Not Solved`, `Infeasible`, `Unbounded`, and `Undefined`:

```
status = plp.LpStatus[model.status]
print(status)
```

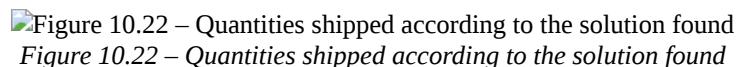
In our case, the state is `Optimal`, therefore an optimal solution has been found for the problem. So, let's see what value the objective function takes on against the solution found:

```
print("Total Cost:", model.objective.value())
```

The value is equal to 2,270,000 euros, which corresponds to the lowest possible cost while satisfying all imposed constraints. If you want to see the solution values of the variables that make up the shipping matrix in a very readable way, you'd better transform them into a pandas dataframe:

```
decision_var_results = np.empty(shape=(n_warehouses * n_countries))
z = 0
for v in model.variables():
    try:
        decision_var_results[z] = v.value()
    z += 1
    except:
        print("error couldn't find value")
decision_var_results = decision_var_results.reshape(n_warehouses,n_countries)
col_idxs = ['Italy','France','Germany','Japan','China','USA']
row_idxs = ['Warehouse ITA','Warehouse DEU','Warehouse JPN','Warehouse USA']
dv_res_df = pd.DataFrame(decision_var_results, columns=col_idxs, index=row_idxs)
dv_res_df
```

The result that comes up is as follows:

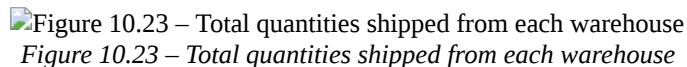
Figure 10.22 – Quantities shipped according to the solution found
Figure 10.22 – Quantities shipped according to the solution found

It is easy to read, for example, that French customers must receive 10,000 units from the German warehouse and 5,000 units from the US warehouse to meet their demand.

If instead you want to check the total quantities shipped from each warehouse, you can run this code:

```
warehouse_shipped_qty = np.zeros(shape=(n_warehouses))
z = 0
for i in range(n_warehouses):
    warehouse_shipped_qty[z] = plp.lpSum(shipping_mtx[i][j].value() for j in range(n_countries)).value()
    z += 1
w_shipped_df = pd.DataFrame(warehouse_shipped_qty, columns=['qty'], index=row_ids)
w_shipped_df
```

You will get this result:

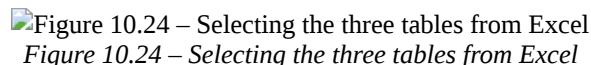


Impressive! You've managed to solve a non-simple linear optimization problem with just a few lines of Python code. Would you have guessed it? Let's now look at how to apply what you learned in Power BI.

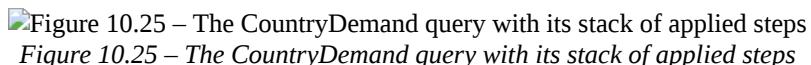
Solving the LP problem in Power BI with Python

Applying what we have just seen in Power BI is not as straightforward as in the other cases we have encountered in the last few chapters. To start, we have to have already loaded the data of country demand, warehouse supply, and shipping costs in the data model, which could arrive from any data source. In our case, we have them ready in Excel, so we will proceed to load them in Power BI Desktop:

1. Click **Excel Workbook** on the ribbon (or **Import data from Excel** in the main canvas), select the `RetailData.xlsx` file in the `Chapter10` folder, and click **Open**.
2. Select the **CountryDemand**, **ShippingCost**, and **WarehouseSupply** tables (the ones with the blue header) and then click **Transform data**:



You are now in the situation described previously. At this point, each of the three queries in Power Query has its own stack of steps that generated it:



In the previous section, you saw that in order to calculate the optimal allocations, *you need to be able to use all three datasets in one Python script*. If you add a Python script as a step in one of the three queries listed previously, you can only interact with the data from that single query in the script. How can you then create a script that can use all the data available? You have to resort to the following stratagem:

In order to use data belonging to more than one query in a Python script, you must first serialize each data structure derived from each of the three queries into a pickle file. Then, create a new query resulting from the merge of the three queries via joins (it is not necessary that the dataset resulting from the joins makes sense, since it will not be used). At this point, add a Python script step for this merge query, in which you deserialize all three previously serialized objects in each query. You will finally have the objects from three different data sources in one script.

This way, you make sure that the Python scripts that serialize the objects of the three datasets are executed first, and then the script that deserializes them is executed in order to solve the linear optimization problem.

Let's see in detail how to accomplish what is described in the previous paragraphs:

1. Select the **CountryDemand** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run Python Script**.

2. Enter the following Python code in the script editor, then click **OK**:

```
import pickle
country_demands = dataset['product_qty'].to_numpy()
pickle.dump( country_demands, open(r"D:\<your-path>\Chapter10\Python\country_demands.pkl", "wb"))
```

3. Click on **Table** under **Value** corresponding to **dataset**. You will always see the country demand data, but behind the scenes, a NumPy vector has been serialized with the name `country_demands.pkl` in the Chapter10/Python folder.

4. Select the **ShippingCost** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run Python Script**.

5. Enter the following Python code in the script editor, then click **OK**:

```
import pickle
n_warehouses = dataset.nunique()['warehouse_name']
n_countries = dataset.nunique()['country_name']
cost_matrix = dataset['shipping_cost'].to_numpy().reshape(n_warehouses, n_countries)
pickle.dump( cost_matrix, open(r"D:\<your-path>\Chapter10\Python\cost_matrix.pkl", "wb"))
```

6. Click on **Table** under **Value** corresponding to **dataset**. You will always see the cost matrix data, but behind the scenes, a NumPy vector has been serialized with the name `cost_matrix.pkl` in the Chapter10/Python folder.

7. Select the **WarehouseSupply** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run Python Script**.

8. Enter the following Python code in the script editor, then click **OK**:

```
import pickle
warehouse_supply = dataset['product_qty'].to_numpy()
pickle.dump( warehouse_supply, open(r"D:\<your-path>\Chapter10\Python\warehouse_supply.pkl", "wb"))
```

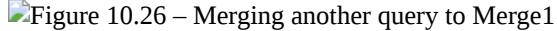
9. Click on **Table** under **Value** corresponding to **dataset**. You will always see the warehouse supply data, but behind the scenes, a NumPy vector has been serialized with the name `warehouse_supply.pkl` in the Chapter10/Python folder.

10. Right-click on the query panel on the left, select **New Query**, then **Combine**, then **Merge Queries as New**.

11. On the **Merge** window, select the **CountryDemand** and **ShippingCost** tables, click on the `country_name` column for both of them, and click **OK**.

12. The new **Merge1** query will appear. Click on it, expand the content of the **ShippingCost** column, and click **OK**.

13. Go to the **Home** tab, make sure that the **Merge1** query is selected, and click on **Merge Queries** in order to also merge **WarehouseSupply** too:


Figure 10.26 – Merging another query to Merge1
Figure 10.26 – Merging another query to Merge1

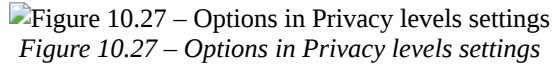
14. Select the **WarehouseSupply** table on the **Merge** window and click the `shippingcost.warehouse_name` and `warehouse_name` columns on both of the tables, then click **OK**.

15. Expand the **WarehouseSupply** column, keep all selected, and click **OK**.

16. Click on the **Transform** tab on the ribbon, click on **Run Python Script**, and enter the script you can find into the `04-linear-optimization-in-power-bi-with-python.py` file in the Chapter10/Python folder.

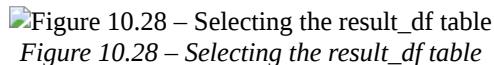
17. You will get the following error: **Formula.Firewall: Query 'Merge1' (step 'Run Python script') references other queries or steps, so it may not directly access a data source. Please rebuild this data combination.**

18. Go to **File** and click **Options and settings** and then **Options**. On the **Options** window, select the **Privacy** tab and click on **Always ignore Privacy Level settings**:



You can do this safely because you are sure that you have set the permissions of all data sources to **Organizational**.

19. Go to the **Home** tab and click on **Refresh Preview**. After a few seconds, click on **Table** for the `result_df` dataframe:



20. You will see the contents of the following table, which shows the values of shipped quantities provided by the solution of the linear optimization problem for each combination of warehouse and country, and the corresponding cost:



21. Click on **Close & Apply**.

Remember that control over source privacy levels is disabled. You should turn it back on to *combine data according to each file's privacy level settings* at the end of this project to be warned in case of possible data leaks due to query folding mechanisms (for more details, see the *References* section).

Important Note

Also remember that if you want to publish your report to the Power BI service, the privacy level of all data sources must be set to **Public**.

Did you know that everything you've seen about LP problems can also be implemented in R? Let's see how to do it.

Solving LP problems with R

If the Python community is very active, certainly the R community is not standing still! In fact, the **Optimization Modeling Package (OMPR)** is available (<https://dirkschumacher.github.io/ompr/>), which is a domain-specific language created to model and solve LP problems in R.

In general, all other packages developed in R that serve the same purpose are mostly matrix-oriented, forcing you to transform all objects into matrices and vectors before passing them to the solver. This task may seem simple enough at first glance, but when the problems to be solved become complex, it may become difficult to write R code to solve them.

The `ompr` package, on the other hand, provides enough expressive power to allow you to model your LP problems incrementally, thanks also to the use of the `%>%` pipe. Therefore, you will feel like you are writing code as if you were using `dplyr` functions, forgetting about matrices and vectors.

In addition, the `ompr` package relies on another package called `ompr.roi` to be able to select the engine of preference for solving LP problems. This package uses, behind the scenes, a sophisticated framework for handling linear and non-linear optimization problems in R called **R Optimization Infrastructure (ROI)**, provided by another package called `ROI` itself (<http://roi.r-forge.r-project.org/>).

In our examples, we'll use the **GLPK** solver added as a plugin by **ROI**.

So, let's see how to set up the LP problem we described in the previous sections in R using the `ompr` package.

Solving the LP problem in R

First, you need to install the packages necessary for the correct functioning of `ompr`. Therefore, follow these steps:

1. Open RStudio and make sure it is referencing your latest CRAN R version (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('ompr')`. Then, press *Enter*.
3. Enter this command: `install.packages('ompr.roi')`. Then, press *Enter*.
4. Enter this command: `install.packages('ROI.plugin.glpk')`. Then, press *Enter*.

You can find all the code that will be shown as follows in the `03-linear-optimization-in-r.R` file in the `chapter10\R` folder.

First, you need to import the needed packages and the data from the Excel `RetailData.xlsx` file that you find in the `Chapter10` folder:

```
library(dplyr)
library(tidyr)
library(readxl)
library(ompr)
library(ompr.roi)
library(ROI.plugin.glpk)
warehouse_supply_tbl = read_xlsx(r'{D:\<your-path>\Chapter10\RetailData.xlsx}', sheet = 'Warehouse Supply')
country_demands_tbl = read_xlsx(r'{D:\<your-path>\Chapter10\RetailData.xlsx}', sheet = 'Country Demands')
cost_matrix_tbl = read_xlsx(r'{D:\<your-path>\Chapter10\RetailData.xlsx}', sheet = 'Shipping Costs')
```

After that, you can compute the arrays and cost matrix from the tibbles, which are needed to then set up the model with `ompr`:

```
n_warehouses <- cost_matrix_tbl %>%
distinct(warehouse_name) %>%
count() %>%
pull(n)
n_countries <- cost_matrix_tbl %>%
distinct(country_name) %>%
count() %>%
pull(n)
warehouse_supply <- warehouse_supply_tbl %>%
pull(product_qty)
country_demands <- country_demands_tbl %>%
pull(product_qty)
cost_matrix <- data.matrix(
cost_matrix_tbl %>%
pivot_wider( names_from = country_name, values_from = shipping_cost ) %>%
select( -warehouse_name )
)
rownames(cost_matrix) <- warehouse_supply_tbl %>% pull(warehouse_name)
```

In order to switch from the vertical form of the cost data in the `cost_matrix_tbl` tibble to the horizontal form, we used the very convenient `pivot_wider()` function provided by the `tidyverse` package.

At this point, defining the model using the functions exposed by `ompr` is almost straightforward if we follow the mathematical model we showed in the *Formulating the LP problem* section:

```
model <- MIPModel() %>%
# define the x integer variables, paying attention to define also the lower bound of 0
add_variable( x[i, j], i = 1:n_warehouses, j = 1:n_countries, type = "integer", lb = 0 ) %>%
# define the objective function, declaring also the "sense" that is the type of problem (minimize)
set_objective( sum_expr(cost_matrix[i, j] * x[i, j], i = 1:n_warehouses, j = 1:n_countries), sense = "Min" )
# add warehouse supply constraints
add_constraint( sum_expr(x[i, j], j = 1:n_countries) <= warehouse_supply[i], i = 1:n_warehouses )
```

```
# add customer demand constraints
add_constraint( sum_expr(x[i, j], i = 1:n_warehouses) >= country_demands[j], j = 1:n_countries )
```

The `sum_expr()` function may seem incomprehensible at first glance. Let's take this piece of code as an example:

```
sum_expr(x[i, j], j = 1:n_countries)
```

It can be read in full as follows: *Take the decision variables $x[i,j]$ obtained by substituting values ranging from 1 to $n_countries$ (that is, 6) for j , then sum the resulting variables together.* In summary, you are asking to calculate this:

```
x[i,1] + x[i,2] + x[i,3] + x[i,4] + x[i,5] + x[i,6]
```

At this point, once the model is defined, you can solve it using the `glpk` solver with this code:

```
result <- model %>%
  solve_model(with_ROI(solver = 'glpk'))
```

The results obtained coincide (obviously, since the solution is optimal) with those already seen using Python's PuLP module:

```
decision_var_results <- matrix(result$solution, nrow = n_warehouses, ncol = n_countries, )
rownames(decision_var_results) <- warehouse_supply_tbl %>% pull(warehouse_name)
colnames(decision_var_results) <- country_demands_tbl %>% pull(country_name)
decision_var_results
```

You will surely recognize the solution already seen before:

	Italy	France	Germany	Japan	China	USA
Warehouse ITA	40000	0	10000	0	0	0
Warehouse DEU	0	10000	15000	5000	0	0
Warehouse JPN	0	0	0	40000	0	0
Warehouse USA	0	5000	0	0	25000	25000

Did you see that you were able to solve an LP problem in R as well? Nothing exceptionally complex, right? Very good!

Let's now apply what we saw in Power BI.

Solving the LP problem in Power BI with R

The implementation complexities of solving our LP problem in Power BI have already been exposed in the *Solving the LP problem in Power BI with Python* section. We will therefore proceed here with the individual steps using R, without dwelling on the details:

1. Open Power BI Desktop and make sure the privacy is set to **Always ignore Privacy Level settings in Options**.
2. Click **Excel Workbook** on the ribbon (or **Import data from Excel** in the main canvas), select the `RetailData.xlsx` file in the `Chapter10` folder, and click **Open**.
3. Select the **CountryDemand**, **ShippingCost**, and **WarehouseSupply** tables (the ones with a blue header) and then click **Transform Data**:

Figure 10.30 – Selecting the three tables from Excel
Figure 10.30 – Selecting the three tables from Excel

4. Select the **CountryDemand** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run R Script**.
5. Enter the following R code in the script editor (change the path properly), then click **OK**:

```

library(dplyr)
country_demands <- dataset %>%
pull(product_qty)
saveRDS(country_demands, r'{D:\<your-path>\Chapter10\R\country_demands.rds}')
country_demand_df <- dataset

```

You will see the country demand data, but behind the scenes, a vector has been serialized with the name `country_demands.rds` in the `Chapter10/R` folder.

6. Select the **ShippingCost** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run R Script**.

7. Enter the following R code in the script editor, then click **OK**:

```

library(dplyr)
library(tidyr)
n_warehouses <- dataset %>%
distinct(warehouse_name) %>%
count() %>%
pull(n)
n_countries <- dataset %>%
distinct(country_name) %>%
count() %>%
pull(n)
cost_matrix <- data.matrix(
dataset %>%
pivot_wider( names_from = country_name, values_from = shipping_cost ) %>%
select( -warehouse_name )
)
rownames(cost_matrix) <- dataset %>%
distinct(warehouse_name) %>%
pull(warehouse_name)
saveRDS(cost_matrix, r'{D:\<your-path>\Chapter10\R\cost_matrix.rds}')
cost_matrix <- dataset

```

You will see the cost matrix data, but behind the scenes, a matrix has been serialized with the name `cost_matrix.rds` in the `Chapter10/R` folder.

8. Select the **WarehouseSupply** query on the left, click on the **Transform** tab on the ribbon, and then click on **Run R Script**.

9. Enter the following R code in the script editor, then click **OK**:

```

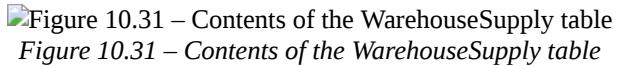
library(dplyr)
warehouse_supply <- dataset %>%
pull(product_qty)
saveRDS(warehouse_supply, r'{D:\<your-path>\Chapter10\R\warehouse_supply.rds}')
warehouse_supply_df <- dataset

```

You will see the warehouse supply data, but behind the scenes, a vector has been serialized with the name `warehouse_supply.rds` in the `Chapter10/R` folder.

10. Right-click on the query panel on the left, select **New Query**, then **Combine**, then **Merge Queries as New**.
11. On the **Merge** window, select the **CountryDemand** and **ShippingCost** tables, then select the `country_name` column for both of them and click **OK**.
12. A new **Merge1** query will appear. Click on it, expand the content of the **ShippingCost** table, and click **OK**.
13. Go to the **Home** tab, make sure that the **Merge1** query is selected, and click on **Merge Queries** in order to also merge **WarehouseSupply** too.
14. Select the **WarehouseSupply** table on the **Merge** window and click the `ShippingCost.warehouse_name` and `warehouse_name` columns on both of the tables, then click **OK**.
15. Expand the **WarehouseSupply** column and click **OK**.
16. Click on the **Transform** tab on the ribbon, click on **Run R Script**, and enter the script, which you can find in the `04-linear-optimization-in-power-bi-with-r.R` file in the `Chapter10\R` folder.

17. You will see the contents of the following table, which shows the values of shipped quantities provided by the solution of the linear optimization problem for each combination of warehouse and country, and the corresponding cost:

Figure 10.31 – Contents of the WarehouseSupply table
Figure 10.31 – Contents of the WarehouseSupply table

18. Click on **Close & Apply**.

Remember that control over source privacy levels is disabled. You should turn it back on to *combine data according to each file's privacy level settings* at the end of this project to be warned in case of possible data leaks due to query folding mechanisms (for more details, see the *References* section).

Summary

In this chapter, you learned how to calculate distances between two geographic points according to the most commonly used definitions in spherical trigonometry using Python and R. You then applied this knowledge to a real-world case in Power BI.

You also learned how to solve the simplest LP problems through some Python and R packages. Again, you applied what you learned to a real-world case in Power BI.

In the next chapter, you'll see how to add some salt to your business analytics thanks to statistics.

References

For additional reading, check out the following books and articles:

- *On Spherical Trigonometry* (<http://www.robingilbert.com/blog/2017-10-01-on-spherical-trigonometry/>)
- *Calculate the Distance Between Two GPS Points with Python (Vincenty's Inverse Formula)* (<https://nathanrooy.github.io/posts/2016-12-18/vincenty-formula-with-python/>)
- *Vectorization and parallelization in Python with NumPy and pandas* (<https://datascience.blog.wzb.eu/2018/02/02/vectorization-and-parallelization-in-python-with-numpy-and-pandas/>)
- *Behind the scenes of the Data Privacy Firewall* (<https://docs.microsoft.com/en-us/power-query/dataprivacyfirewall>)

11 Adding Statistics Insights: Associations

As you saw in the previous chapter, enriching your data can be done through the application of complex algorithms. In addition to distances and linear programming, statistics can often be the ultimate weapon for data analysis. We will explain the basic concepts of some statistical procedures that aim to extract relevant insights about the associations between variables from your data.

In this chapter, you will learn about the following topics:

- Exploring associations between variables
- Correlation between numeric variables
- Correlation between categorical and numeric variables

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

Exploring associations between variables

At first glance, you might wonder what the point of finding relationships between variables is. Being able to understand the behavior of a pair of variables and identify a pattern in their behavior helps business owners identify key factors to divert certain indicators of company health to their benefit. Knowing the pattern that binds the trend of two variables gives you the power to predict with some certainty one of them by knowing the other. So, knowing the tools to uncover these patterns gives you a kind of analytical superpower that is always appealing to business owners.

In general, two variables are **associated** with each other when the values of one of them are in some way related to the values of the other. When you can somehow measure the extent of the association of two variables, it is called **correlation**. The concept of correlation is immediately applicable in a case where the two variables are numerical. Let's see how.

Correlation between numeric variables

The first thing we generally do to understand whether there is an association between two numeric variables is to represent them on the two Cartesian axes in order to obtain a **scatterplot**:

A simple scatterplot

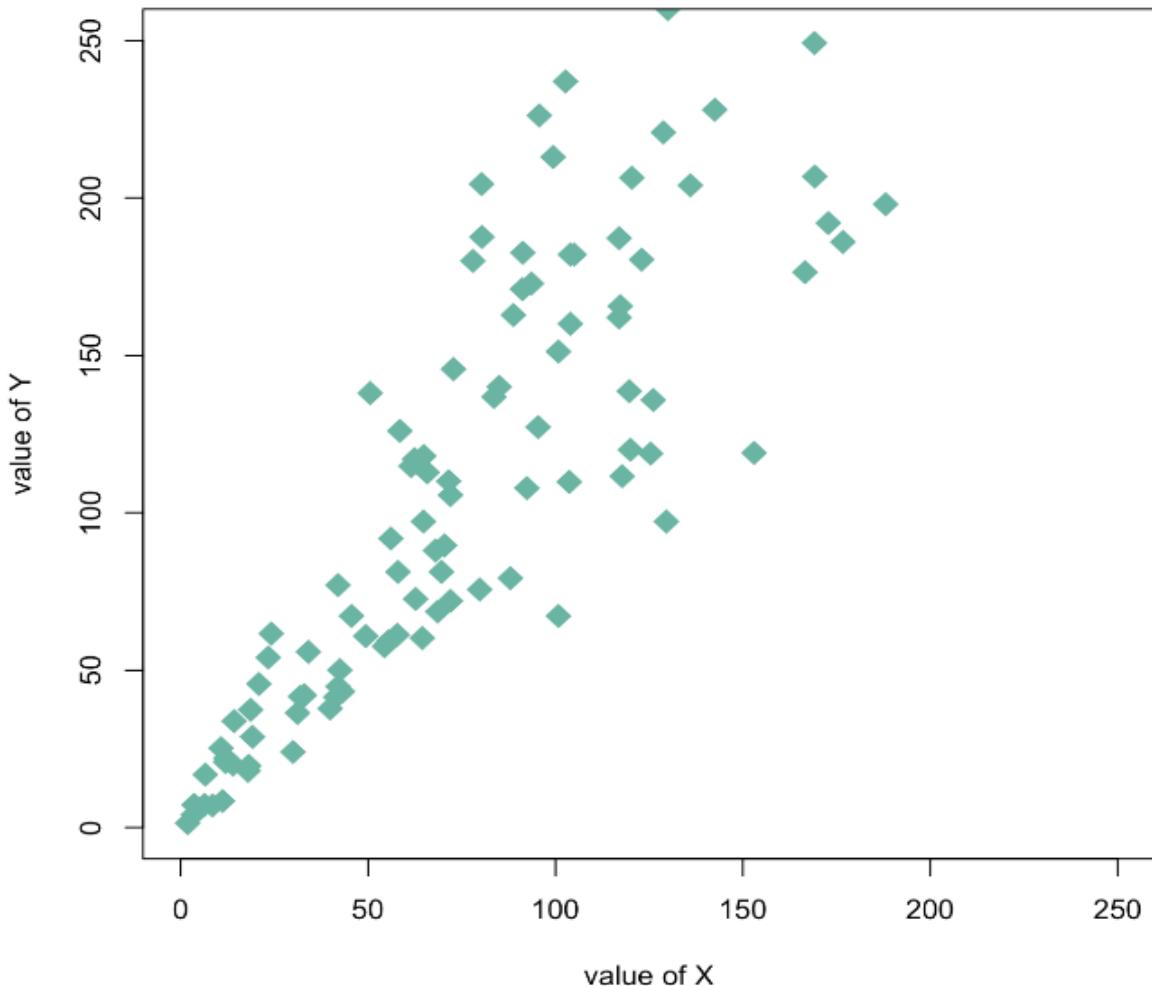
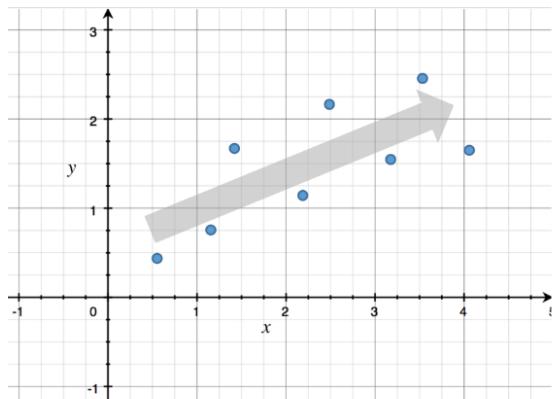


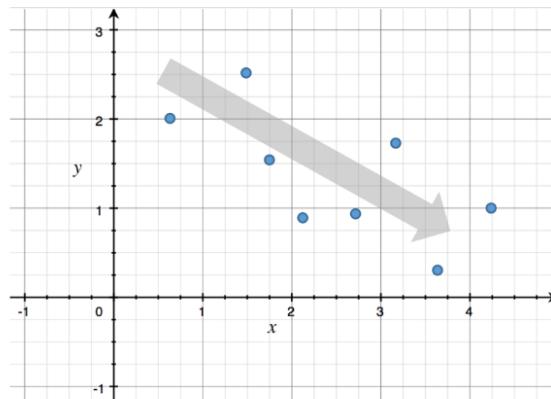
Figure 11.1 – A simple scatterplot

Using a scatterplot, it is possible to identify three important characteristics of a possible association:

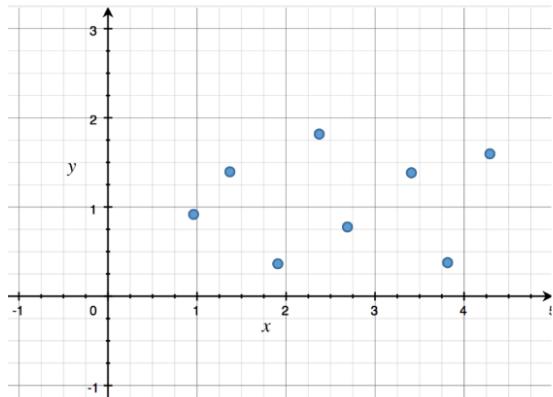
- **Direction:** It can be *positive* (increasing), *negative* (decreasing), or *not defined* (no association found or increasing and decreasing). If the increment of a variable corresponds to the increment of the other, the direction is positive; otherwise, it is negative:



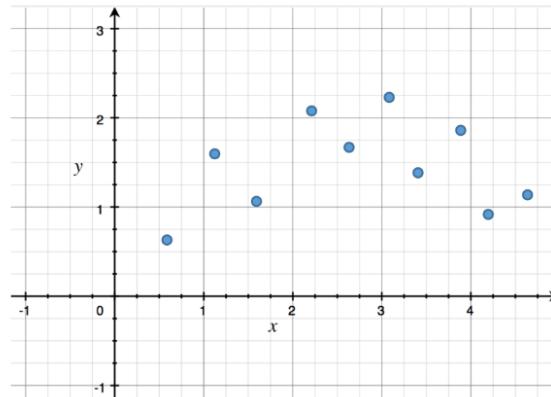
positive direction



negative direction



not defined
no association



not defined
increasing & decreasing

Figure 11.2 – Direction types of the association

- **Form:** It describes the general form that association takes in its simplest sense. Obviously, there are many possible forms, but there are some that are more common, such as *linear* and *curvilinear* (nonlinear) forms:

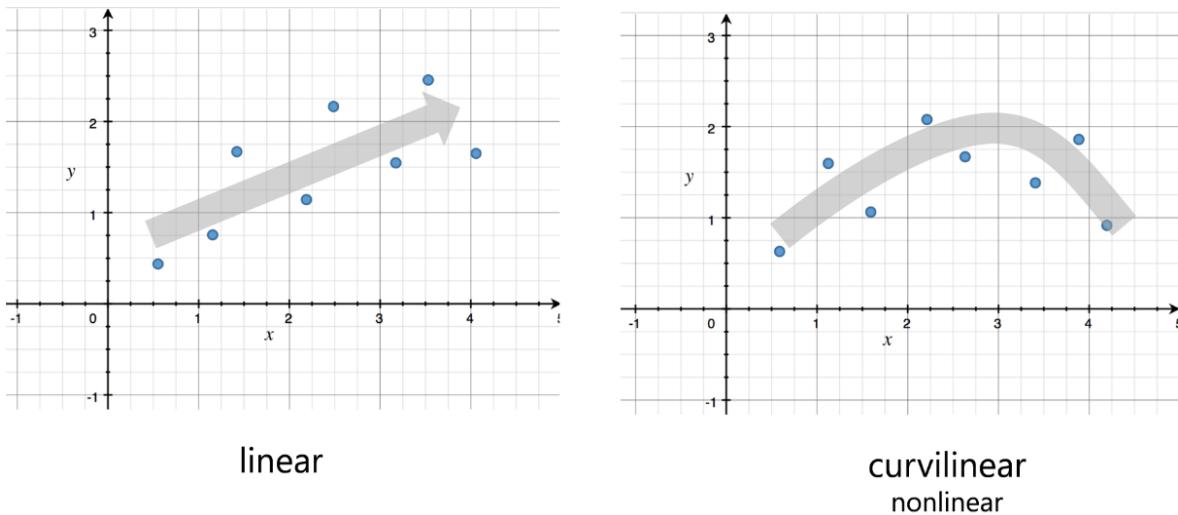


Figure 11.3 – Shapes of the association

- **Strength:** The strength of an association is determined by how close the points in the scatterplot follow the line that draws the generic shape of the association.

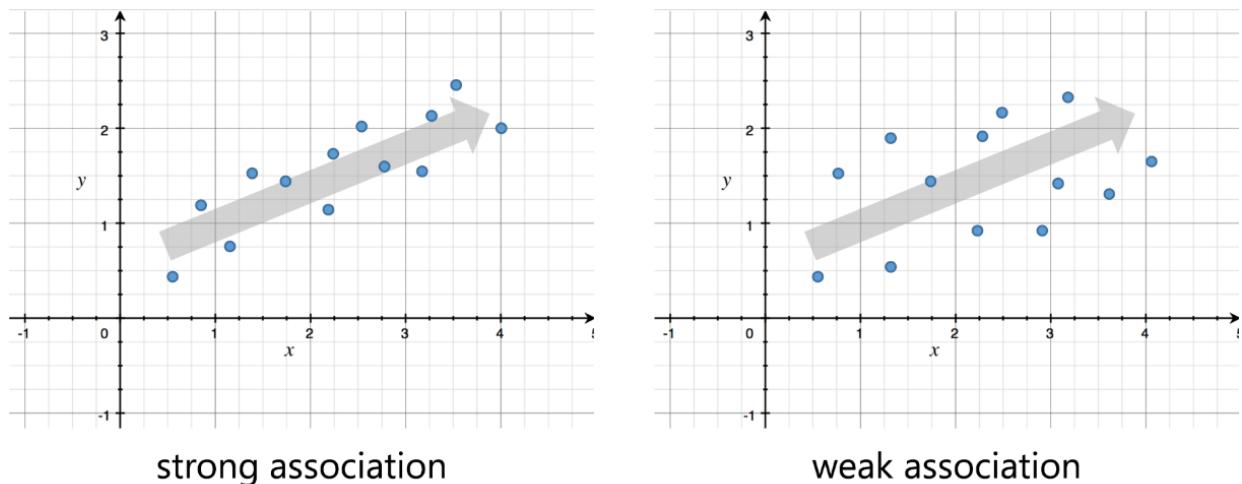


Figure 11.4 – Strength of the association

As these visual patterns become measurable through the application of mathematical concepts, we can define different types of correlations between numeric variables.

Karl Pearson's correlation coefficient

Pearson's correlation coefficient (r) measures the degree of linearity of the association between the variables under analysis. This means that if $r = 1$ or $r = -1$, then the two variables are in a *perfect linear relationship*. The sign of the coefficient determines the direction of the association. If instead r is close to 0 indifferently on the negative or positive side, it means that the association between the variables is *very weak*. The coefficient cannot take values outside the range $[-1, 1]$.

The square of Pearson's coefficient (R^2) is called the **coefficient of determination** and measures the percentage of **variance** (which measures the dispersion of observations from their mean value) in one variable due to the variance of the other variable, assuming the association is linear. For example, if the correlation coefficient r

between a person's weight and height variables measures 0.45, it can be said that about 20% (0.45×0.45) of the change (variance) in a person's weight is due to the change in their height.

Calculating the correlation coefficient r is rarely done by hand, as any data management platform provides a function that easily calculates it. If you are curious, given a dataset of n entities, identify the variables x and y for which you want to calculate the correlation coefficient – this is the formula that allows you to calculate it:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Figure 11.5 – Formula for the Pearson correlation coefficient

The \bar{x} and \bar{y} values correspond to the mean values of the homonymous variables in the dataset. Keep in mind that Pearson's correlation function is **symmetric**, meaning that the order of the columns for which it is calculated does not matter.

Examples of the correlation coefficient r calculated for some specific associations (**Boigelot distributions**) are shown in *Figure 11.6*:

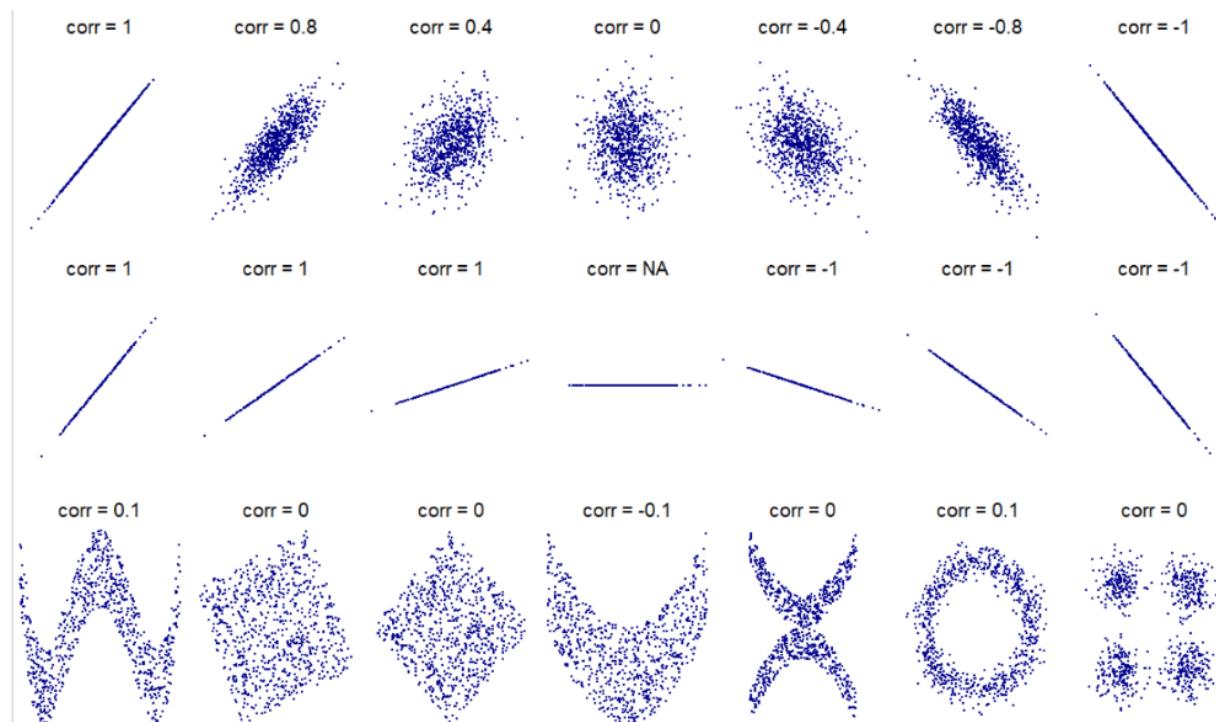


Figure 11.6 – Pearson's correlation calculated over Boigelot distributions

The first row of scatterplots in *Figure 11.6* gives an idea of how the magnitude of the correlation measures the strength of the association tending toward a linear relationship.

The second row shows that, regardless of the angle of the linear relationship, the correlation coefficient r is always equal to 1 in absolute value, that is, it always correctly identifies the linear relationship and its direction. The only exception is the case of the horizontal linear relationship at the center, for which the coefficient r is undefined, since for all values of x there is a single value of y .

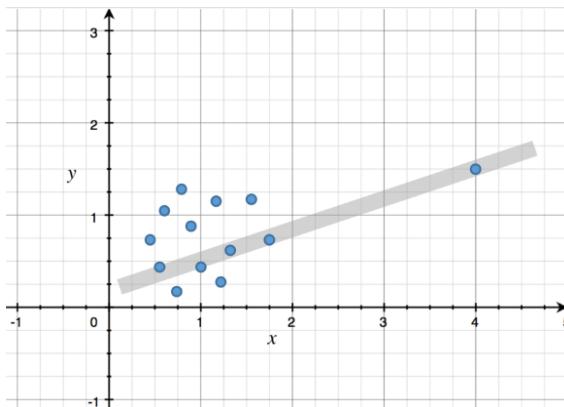
The third row shows one of the most important limitations of the correlation coefficient:

Important Note

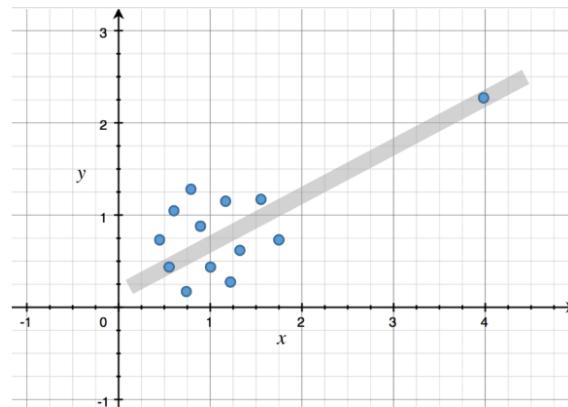
Pearson's correlation coefficient is not capable of detecting a pattern in an association between two variables as much as the latter are nonlinear.

These are the other limitations:

- With a very small dataset size (say, 3-6 observations), it might appear that an association is present even though it does not exist.
- The correlation coefficient is very sensitive to outliers (observations located far from most others). It can also happen that outliers give a false idea of the existence of association:



wrong correlation

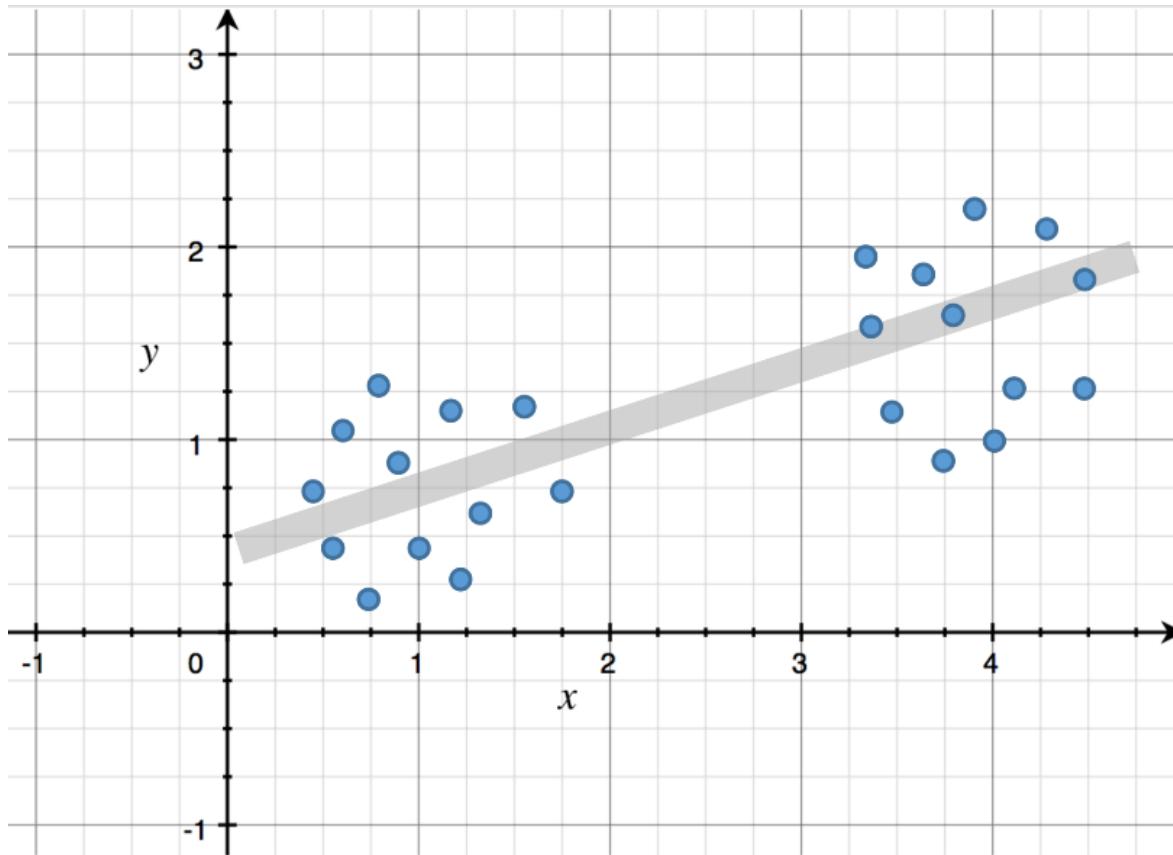


wrong correlation

Figure 11.7 – Wrong correlation values due to outliers

We will cover outliers in more detail in *Chapter 12, Adding Statistics Insights, Outliers, and Missing Values*.

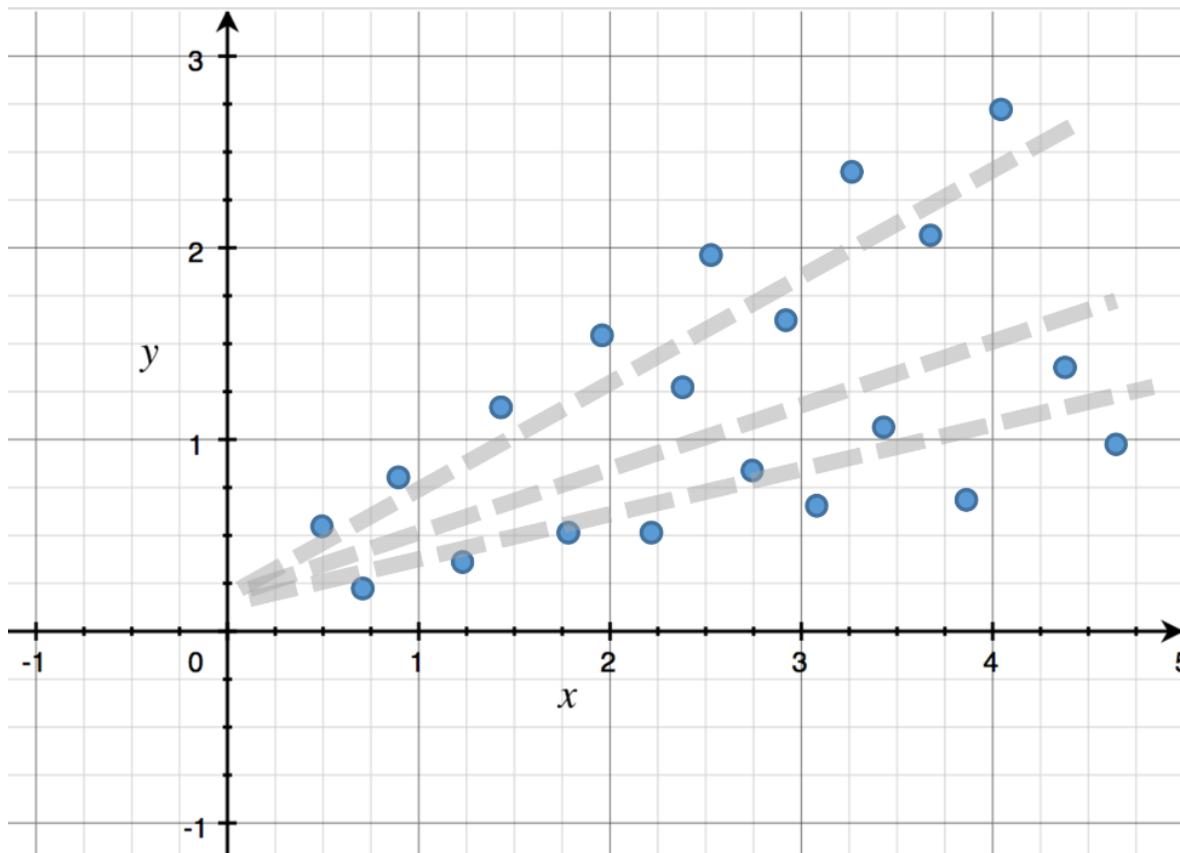
- If the observations in the dataset are divided into different clusters, this may induce a false sense of association:



wrong correlation due to clusters

Figure 11.8 – Wrong correlation value due to clusters

- The variables used in the calculation of the correlation coefficient must be defined on a continuous scale. For variables based on a discrete scale, such as the 1-10 rating of a service, you must use Spearman's rank correlation, which we will look at in the next section.
- If in an association there is a variable that has unequal variability with a range of values of a second variable, we are faced with a case of **heteroscedasticity**. The scatterplot assumes the typical shape of a cone, like the one in *Figure 11.1*. In this case, the correlation coefficient could identify an incorrect linear relationship (more than one linear relationship would satisfy the conic form of the scatterplot):



wrong correlation due to heteroscedasticity

Figure 11.9 – Wrong correlation value due to heteroscedasticity

- The variables involved in calculating the Pearson correlation coefficient should not be highly **skewed**, that is, the distribution of the variables must not be distorted or asymmetrical with respect to a symmetrical bell curve (normal distribution, where the mean, median, and mode are equal), otherwise a reduction in the true size of the correlation magnitude could occur:

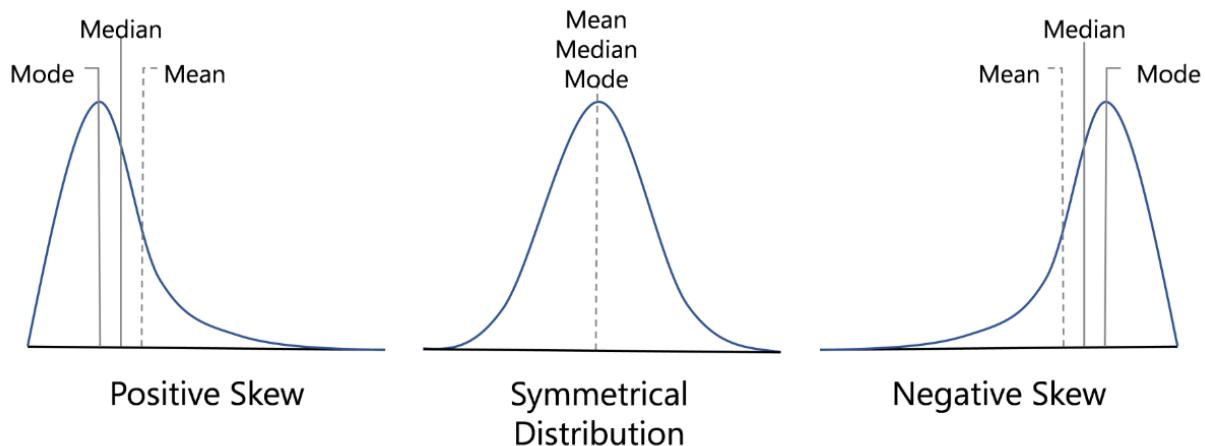


Figure 11.10 – Distribution skewness types

Having clarified these limitations, the first question that comes up is: how can I calculate the correlation between two numeric variables when the association is nonlinear, when the variables are based on an ordinal scale, or when they have a skewed distribution? One of the possible solutions to this problem was provided by Charles Spearman.

Charles Spearman's correlation coefficient

Charles Spearman introduced a nonparametric alternative to Pearson's correlation. A **nonparametric statistical method** refers to the fact that it does not assume that the data on which it is calculated follows specific models described by a handful of parameters. This method is summarized in a new correlation coefficient called **Spearman's rank-order correlation coefficient** (denoted by ρ , *rho*). Spearman's coefficient measures the strength and direction of the association between two variables once their observations have been ranked according to their value. The formula that calculates it is as follows:

$$\rho = 1 - \frac{6 \sum D^2}{n(n^2 - 1)}$$

Figure 11.11 – Spearman's rank-order correlation coefficient formula

The value D is the difference between the two ranks of each observation. To learn more details about the calculation, take a look at the references. Keep in mind that Spearman's correlation function is also symmetric.

Spearman's correlation coefficient ranges from -1 to +1. The sign of the coefficient indicates whether it is a positive or negative monotone association.

Important Features of Spearman's Correlation

Because Spearman's correlation applies to ranks, it provides a measure of a **monotonic association** between two continuous random variables and is the best-fitting correlation coefficient for ordinal variables. Because of the way it is calculated, as opposed to Pearson's, Spearman's correlation is *robust to outliers*.

Remember that an association is said to be monotonic if it is increasing over its entire domain or decreasing over its entire domain (not a combination of the two):

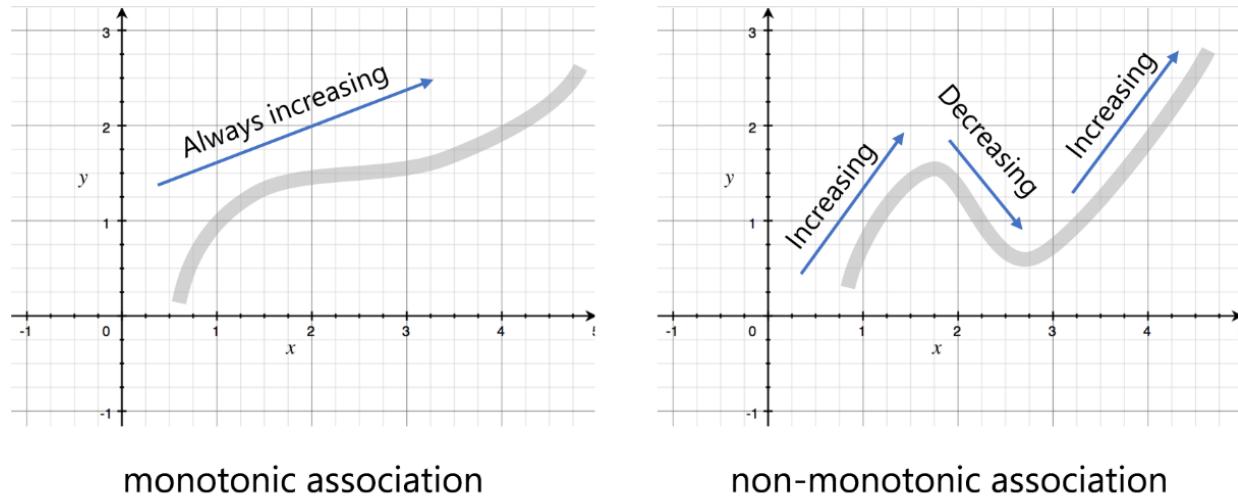


Figure 11.12 – Monotonic and non-monotonic associations

Important Note

It's important to check the monotonicity of the association, since in general the correlation coefficients are not able to accurately describe non-monotonic relationships.

Calculating the Spearman correlation coefficient for the Boigelot distributions, we get this:

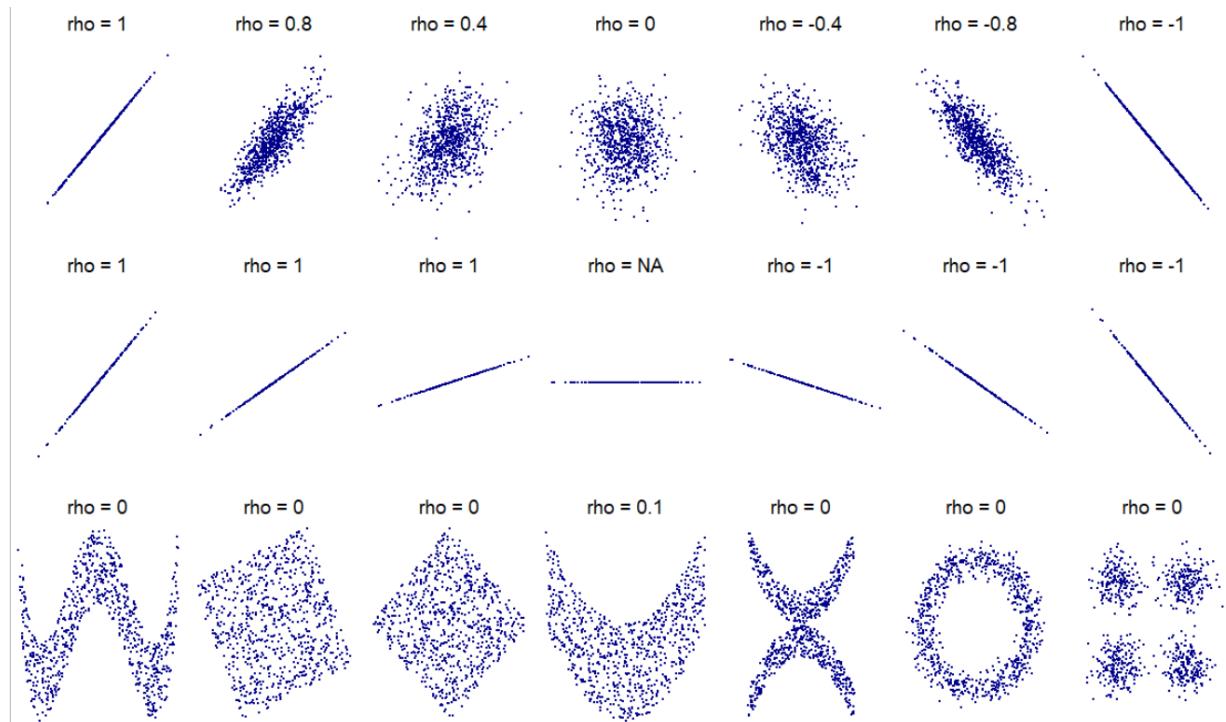


Figure 11.13 – Spearman’s correlation calculated over Boigelot distributions

Even Spearman’s correlation fails to capture the nonlinearity you observe in the third row of distributions in *Figure 11.13*. And this is due not to the nonlinearity of the above associations, but to their non-monotonicity. In contrast, for monotonic nonlinear associations, Spearman’s correlation is better suited than Pearson’s:

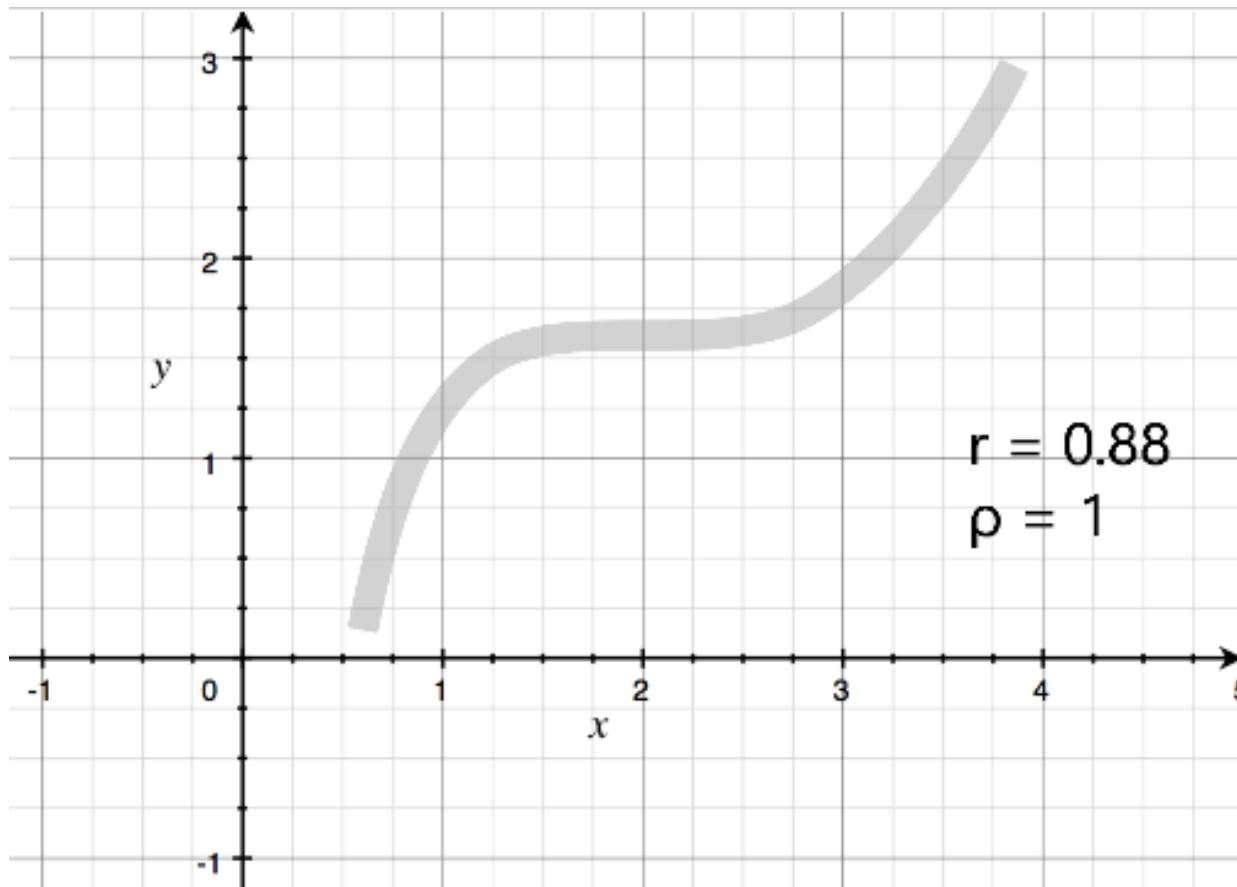


Figure 11.14 – Pearson's and Spearman's correlation over a nonlinear monotonic association

Spearman's correlation is not the only one that uses data ranking in its calculation. Kendall's also uses this strategy. Let's take a look at its features.

Maurice Kendall's correlation coefficient

Kendall's rank correlation coefficient (τ , tau) is also a nonparametric method of detecting associations between two variables. Its calculation is based on the concept of **concordant pairs** and **discordant pairs** (check the *References* section for more details). The formula useful in calculating Kendall's correlation coefficient is as follows:

$$\tau = \frac{2(n_c - n_d)}{n(n - 1)}$$

Figure 11.15 – Kendall's correlation coefficient formula

The n_c value represents the number of concordant pairs, and the n_d value represents the number of discordant pairs. Like the other correlation functions, Kendall's is symmetric too.

All of the assumptions made for Spearman's correlation also apply to Kendall's. Here is a comparison between the two correlations:

- Both correlations handle ordinal data and nonlinear continuous monotonic data very well.
- Both correlations are robust to outliers.
- Kendall correlation is preferred to Spearman correlation when the sample size is small and when it has many tied ranks.
- Kendall's coefficient is usually smaller than Spearman's.
- Kendall's correlation has more computational complexity than Spearman's.

That said, let's use all three correlation coefficients in a real case.

Description of a real case

Your boss has asked you to carry out a world population analysis requested by a client. Specifically, you need to understand whether there is a relationship and of what magnitude between life expectancy and per capita gross domestic product (GDP) over the years.

Looking around the web for useful data for the purpose, you realize that there is a portal that can help your case named **Gapminder** (<https://www.gapminder.org/>). It fights devastating misconceptions and promotes a fact-based worldview everyone can understand combining data from multiple sources into unique coherent time series that can't be found elsewhere. The portal precisely exposes data on life expectancy (<http://bit.ly/life-expectancy-data>) and data on GDP per capita (<http://bit.ly/gdp-per-capita-data>). In addition, there is someone who has transformed the data to collect it in a more suitable form for our purposes and has shared his work with the community in a CSV file (<http://bit.ly/gdp-life-expect-data>).

Awesome! You have everything you need to start your analysis in Python.

Implementing correlation coefficients in Python

To run the code in this section, you need to install the Seaborn module in your `pbi_powerquery_env` environment. As you've probably learned by now, proceed as follows:

1. Open the Anaconda prompt.
2. Enter the command `conda activate pbi_powerquery_env`.

3. Enter the command `pip install seaborn`.

The code you'll find in this section is available in the file `01-gdp-life-expectancy-analysis-in-python.py` in the `Chapter11\Python` folder.

At this point, let's take a quick look at the data in the above CSV file, while also importing the forms needed for the rest of the operations:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
dataset_url = 'http://bit.ly/gdp-life-expect-data'
df = pd.read_csv(dataset_url)
df.head()
# If you're not using VS Code run this instead
# print(df.head())
```

You'll see something like this:

	country	year	pop	continent	lifeExp	gdpPercap
0	Afghanistan	1952	8425333.0	Asia	28.801	779.445314
1	Afghanistan	1957	9240934.0	Asia	30.332	820.853030
2	Afghanistan	1962	10267083.0	Asia	31.997	853.100710
3	Afghanistan	1967	11537966.0	Asia	34.020	836.197138
4	Afghanistan	1972	13079460.0	Asia	36.088	739.981106

Figure 11.16 – A sample of the dataset about GDP and life expectancy

The variables we are interested in are `lifeExp` and `gdpPercap`. Before drawing a scatterplot of the two, let's also take a look at the distribution of each of them. So let's define the functions we will use for the plots and draw the distributions:

```
def distPlot(data, var, title, xlab, ylab, bins=100):
    hplot = sb.distplot(data[var], kde=False, bins=bins)
    plt.title(title, fontsize=18)
    plt.xlabel(xlab, fontsize=16)
    plt.ylabel(ylab, fontsize=16)

    return hplot
def scatterPlot(data, varx, vary, title, xlab, ylab):
    hplot = sb.scatterplot(varx, vary, data=data)
    plt.title(title, fontsize=18)
    plt.xlabel(xlab, fontsize=16)
    plt.ylabel(ylab, fontsize=16)

    return hplot
distPlot(data=df, var='lifeExp', title='Life Expectancy', xlab='Life Expectancy years', ylab='Frequency')
# In case you're not using a Jupyter notebook run also the following:
# plt.show()
distPlot(data=df, var='gdpPercap', title='GDP / capita', xlab='GDP / capita ($)', ylab='Frequency')
# In case you're not using a Jupyter notebook run also the following:
# plt.show()
```

The plots created are as follows:

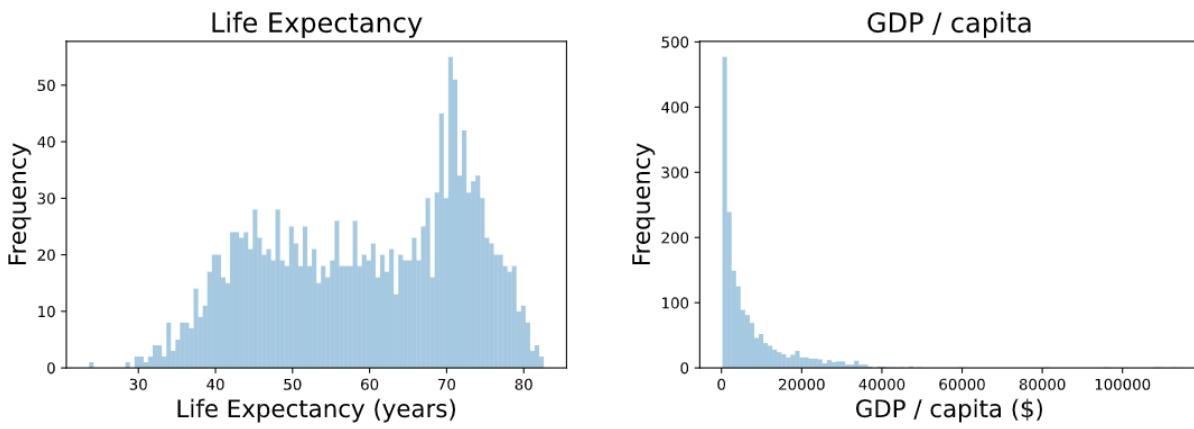


Figure 11.17 – Distributions of life expectancy and GDP variables

As you can see, while the life expectancy distribution more or less approximates the normal distribution, the GDP distribution is totally positive-skewed. Should there be a significant association between the two variables, you should then expect a likely non-linear scatter plot. Let's check it out:

```
scatterPlot(data=df, varx='lifeExp', vary='gdpPercap', title='Life Expectancy vs GDP/Capita', xlabel='Life Expectancy (years)', ylabel='GDP/Capita ($)', color='blue', size=10, alpha=0.5)
# In case you're not using a Jupyter notebook run also the following:
# plt.show()
```

You get this:

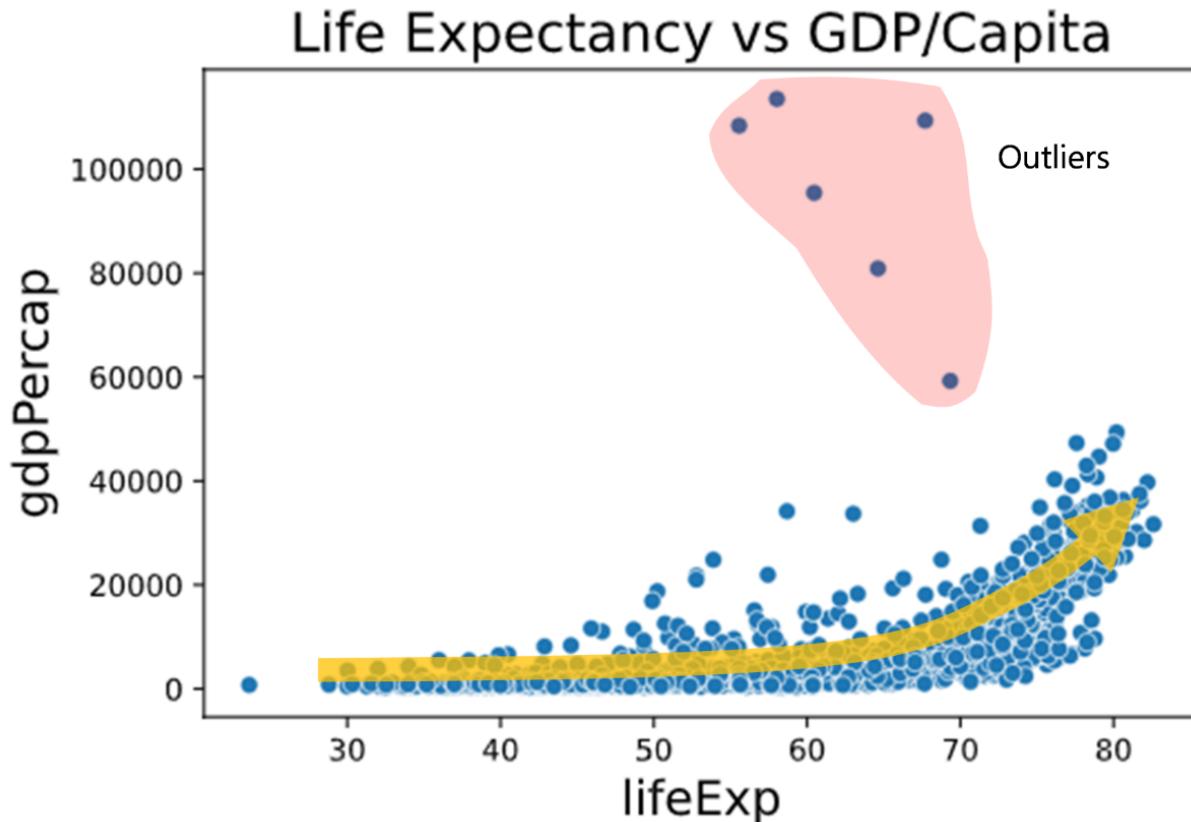


Figure 11.18 – Scatterplot between life expectancy and GDP per capita

From *Figure 11.18*, you can see that the association between the two variables is obvious and, as expected, is nonlinear (background arrow). What is more, outliers are present that are highlighted at the top. You are therefore faced with two assumptions that invalidate Pearson's correlation. Fortunately, the association is monotonically increasing, so the Spearman and Kendall correlations should detect the pattern more accurately.

The Python implementation of the three correlation coefficients is already made available by pandas. Therefore, correlation analysis is straightforward:

```
df[['lifeExp', 'gdpPercap']].corr(method='pearson')
df[['lifeExp', 'gdpPercap']].corr(method='spearman')
df[['lifeExp', 'gdpPercap']].corr(method='kendall')
```

The `corr()` function of a pandas dataframe returns the correlation calculation for each pair of its numeric columns. Since the correlation functions are symmetric (that is, the order of the columns for which it is computed does not matter), the three calls return three dataframes each having all the numeric features as rows and columns, and the correlations between the features as values:

Pearson		Spearman		Kendall	
		lifeExp	gdpPercap	lifeExp	gdpPercap
lifeExp	1.000000	0.583706		lifeExp	1.000000
gdpPercap	0.583706	1.000000		gdpPercap	0.826471

Figure 11.19 – Correlations between life expectancy and GDP per capita

As you can see, all three correlations identify a positive association between the two variables. As expected, the strength detected by Pearson's correlation is the weakest ($r = 0.58$). In contrast, the Spearman and Kendall correlations have a higher magnitude, $\rho = 0.83$ and $\tau = 0.64$, respectively. Especially from Spearman's correlation, it is evident that the two variables are strongly correlated.

It often happens that in a data science project, you have to select the most predictive variables towards a target variable to predict when you have a large number of columns. The technique of correlation can certainly help us in this: assuming that the target variable is GDP per capita, we could decide to keep all those variables as predictors that have a correlation with the target variable greater than 0.7. You understand that in this case, it is important to consider the right correlation method, otherwise, you would risk rejecting a variable that is strongly correlated to the target one.

Great! Let's see how to replicate this analysis using R as well.

Implementing correlation coefficients in R

One of the packages we recommend using in R for calculating correlation coefficients is **corrr** (<https://github.com/tidymodels/corrr>). It allows you to explore and rearrange the tibbles returned by the `correlate()` function very easily according to the practices suggested by Tidyverse. Therefore, you need to install the `corrr` package in the most recent version of CRAN R you have:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('corrr')`. Then press *Enter*.

The code you'll find in this section is available in the file `01-gdp-life-expectancy-analysis-in-r.R` in the `chapter11\R` folder.

So let's proceed to import the necessary libraries, load the data from the CSV file on the web into a tibble, and display the first few lines:

```
library(readr)
library(dplyr)
```

```

library(corr)
library(ggplot2)
dataset_url <- 'http://bit.ly/gdp-life-expect-data'
tbl <- read_csv(dataset_url)
tbl

```

You will see this in the console:

```

# A tibble: 1,704 x 6
  country      year    pop continent lifeExp gdpPercap
  <chr>     <dbl>   <dbl>   <chr>      <dbl>     <dbl>
1 Afghanistan 1952 8425333 Asia       28.8     779.
2 Afghanistan 1957 9240934 Asia       30.3     821.
3 Afghanistan 1962 10267083 Asia      32.0     853.
4 Afghanistan 1967 11537966 Asia      34.0     836.
5 Afghanistan 1972 13079460 Asia      36.1     740.
6 Afghanistan 1977 14880372 Asia      38.4     786.
7 Afghanistan 1982 12881816 Asia      39.9     978.
8 Afghanistan 1987 13867957 Asia      40.8     852.
9 Afghanistan 1992 16317921 Asia      41.7     649.
10 Afghanistan 1997 22227415 Asia     41.8     635.
# ... with 1,694 more rows
> |

```

Figure 11.20 – First rows of the population tibble

Also, in this case, we define the functions necessary to draw a distribution plot and a scatterplot and use them to generate the plots of the distributions of the variables `lifeExp` and `gdpPercap`:

```

distPlot <- function(data, var, title, xlab, ylab, bins=100) {
  p <- ggplot( data=data, aes_string(x=var) ) +
    geom_histogram( bins=bins, fill="royalblue3", color="steelblue1", alpha=0.9 ) +
    ggtitle(title) + xlab(xlab) + ylab(ylab) +
    theme( plot.title = element_text(size=20), axis.title = element_text(size=16) )
  return(p)
}
scatterPlot <- function(data, varx, vary, title, xlab, ylab) {
  p <- ggplot( data=data, aes_string(x=varx, y=vary)) +
    geom_point(
      color='steelblue1', fill='royalblue3',
      shape=21, alpha=0.8, size=3
    ) +
    ggtitle(title) + xlab(xlab) + ylab(ylab) +
    theme( plot.title = element_text(size=20), axis.title = element_text(size=16) )
  return(p)
}
distPlot(data = tbl, var = 'lifeExp', title = 'Life Expectancy', xlab = 'Life Expectancy')
distPlot(data = tbl, var = 'gdpPercap', title = 'GDP / capita', xlab = 'GDP / capita ($)', ylab =

```

These are the plots you get:

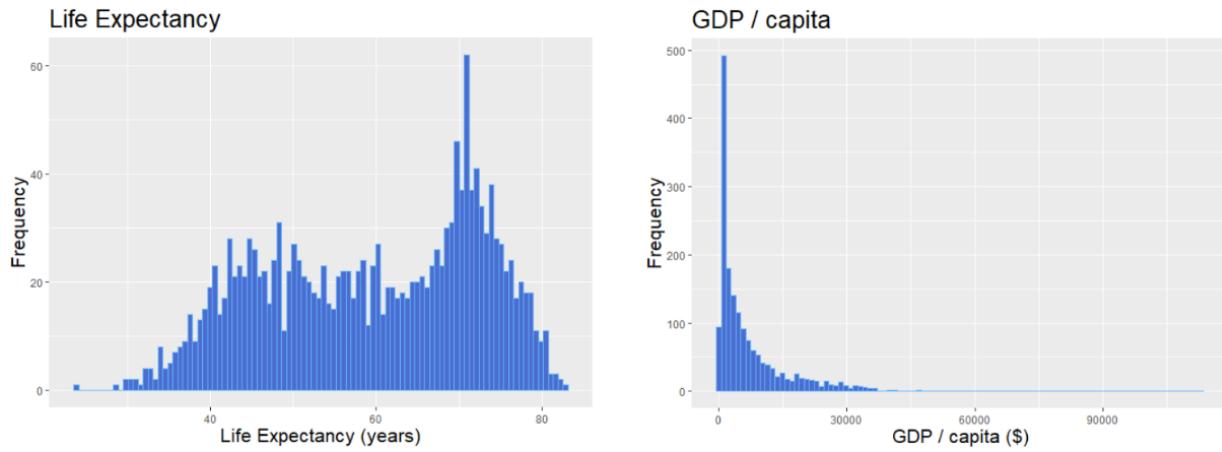


Figure 11.21 – Distribution plots of Life Expectancy and GDP per capita

You can get the scatterplot between the two variables as follows:

```
scatterPlot(data = tbl, varx = 'lifeExp', vary = 'gdpPercap', title = 'Life Expectancy vs GDP/Capita')
```

This is the graph you get:

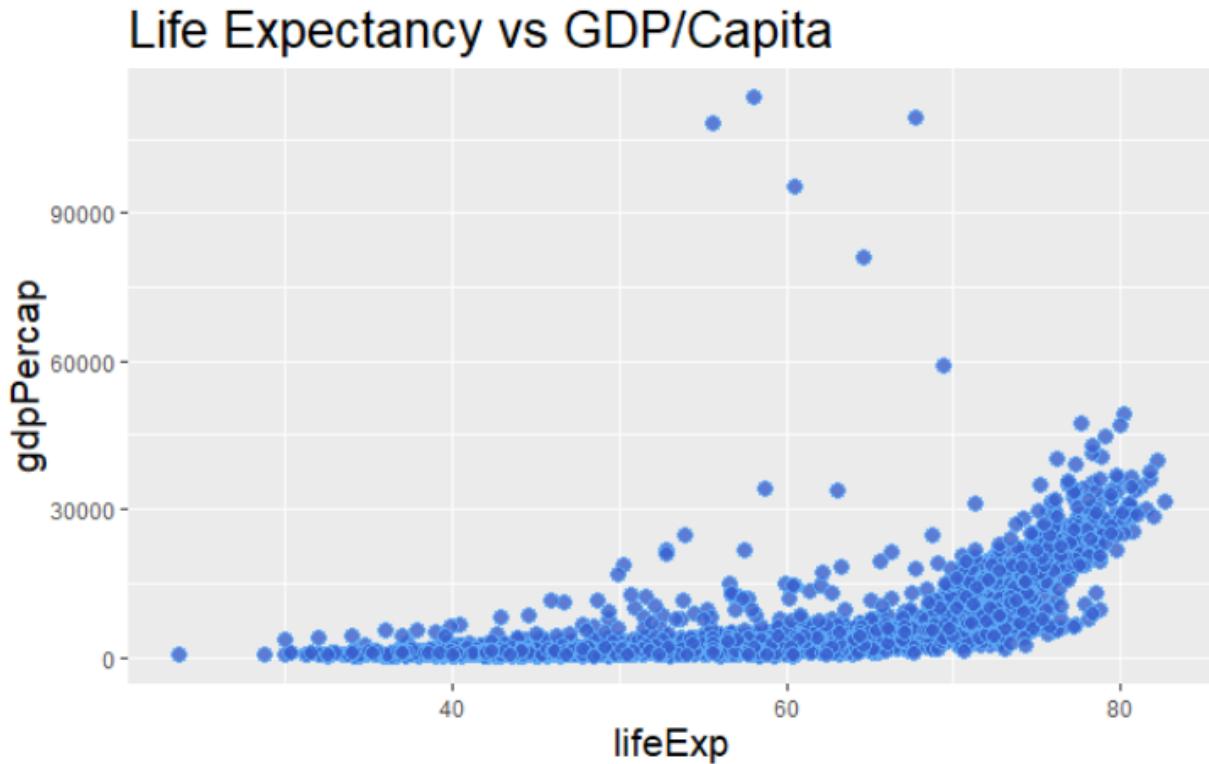


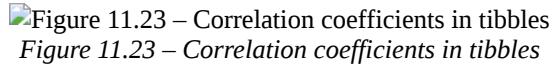
Figure 11.22 – Scatterplot between Life Expectancy and GDP per capita

Correlation matrices (persisted in tibbles) are obtained straightforwardly via the `corr` package's `correlate()` function as follows:

```
tbl %>% select( lifeExp, gdpPercap ) %>% correlate( method = 'pearson' )
tbl %>% select( lifeExp, gdpPercap ) %>% correlate( method = 'spearman' )
```

```
tbl %>% select( lifeExp, gdpPercap ) %>% correlate( method = 'kendall' )
```

These are the console results:



Pretty simple, right!? Now that you know how to get correlation coefficients in both Python and R, let's implement what you learned in Power BI.

Implementing correlation coefficients in Power BI with Python and R

Power BI has the ability to introduce a minimum of statistical analysis for the data that has been loaded into the data model thanks to DAX. You can find a list of statistical functions you can use at this link:

<http://bit.ly/dax-stats-func>. As for the simple *Pearson correlation*, you can use it for columns in an already loaded table thanks to the predefined quick measures, which behind the scenes add some sometimes non-trivial DAX code for you. For more details, you can click on this link: <http://bit.ly/power-bi-corr-coef>. However, there is no easy way to implement the Spearman and Kendall correlation coefficients.

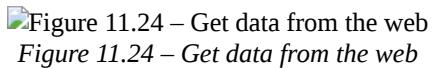
Also, you have probably heard of the **correlation plot** available in Microsoft's AppSource (<https://bit.ly/power-bi-corr-plot>). You might think it would be possible to use that to get the correlation coefficients. First of all, the plot only displays Pearson correlation coefficients and not Spearman and Kendall ones. Also, you cannot extract coefficients from a visual or custom visual to persist them in a table in order to use them for later calculations (for example, feature selection). So the correlation plot idea is totally off the mark in this case.

The only way to proceed is to use Python or R. Let's see how to do that.

In this case, due to the simplicity of the code, we will implement the correlation coefficients in both Python and R in one project.

First, make sure that Power BI Desktop references the correct versions of Python and R in **Options**. Then follow these steps:

1. Click on **Get Data**, select **Web**, and click on **Connect**:



2. Enter the `http://bit.ly/gdp-life-expect-data` string into the **URL** textbox and click **OK**.
3. You'll see a preview of the data. Then click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.

5. Enter the following Python script and then click **OK**:

```
import pandas as pd
corr_df = dataset.corr(method='pearson')
# You need to convert row names into a column
# in order to make it visible in Power BI
corr_df.index.name = 'rowname'
corr_df.reset_index(inplace=True)
```

You can find this code in the file `02-gdp-life-expectancy-analysis-in-power-bi-with-python.py` in the `Chapter11\Python` folder.

6. We are only interested in the data in `corr_df`. So, click on its **Table** value.
7. You'll see the preview of the Pearson correlation coefficients between all the numeric columns of the dataset.
8. Click **Home** on the ribbon and then click **Close & Apply**.

9. Repeat steps 1 to 3.
10. Click **Transform** on the ribbon and then **Run R script**.
11. Enter the following R script and then click **OK**:

```
library(dplyr)
library(corr)
# You need to select only numeric columns
# in order to make correlate() work
corr_tbl <- tbl %>%
  select( where(is.numeric) ) %>%
  correlate( method = 'spearman' )
```

You can find this code in the file `02-gdp-life-expectancy-analysis-in-power-bi-with-r.R` in the `Chapter11\R` folder.

 12. We are only interested in the data in `corr_tbl`. So, click its **Table** value.
 13. You'll see the preview of the Spearman correlation coefficients between all the numeric columns of the dataset.
 14. Click **Home** on the ribbon and then click **Close & Apply**.

Awesome! You have just calculated the correlation coefficients according to Pearson and Spearman for the numeric columns of a source dataset in Power BI with Python and R. Simple, isn't it?

You're probably wondering, *Okay, all clear on the numeric variables. What if I had categorical (non-numeric) variables? How can I calculate the correlation between them? And what about the correlation between a numeric variable and a categorical one?* Let's take a look at how to approach this type of analysis.

Correlation between categorical and numeric variables

We have shown that, in the case of two numeric variables, you can get a sense of the association between them by taking a look at their scatterplot. Clearly, this strategy cannot be used when one or both variables are categorical. Note that a variable is **categorical** (or qualitative, or nominal) when it takes on values that are names or labels. For example, smartphone operating systems (iOS, Android, Linux, and so on).

Let's see how to analyze the case of two categorical variables.

Considering both variables categorical

So, is there a graphical representation that helps us understand whether there is a significant association between two categorical variables? The answer is yes and its name is a **mosaic plot**. In this section, we will take the Titanic disaster dataset as a reference dataset. In order to have an idea of what a mosaic plot looks like, let's take into consideration the variables `Survived` (which takes values `1` and `0`) and `Pclass` (*passenger class*, which takes values `1`, `2`, and `3`). Since we want to study the association between these two variables, we consider the following mosaic plot generated by them:

Figure 11.25 – Mosaic plot of the variables Survived and Pclass
Figure 11.25 – Mosaic plot of the variables Survived and Pclass

In short, the objective of the mosaic plot is to provide, at a glance, the strength of the association between the individual elements of each variable through the color of the tiles that represent the pairs of elements in question.

Important Note

Basically, the more the color of the tile tends toward *dark blue*, the more there are observations represented by that tile compared to the expected quantity in the case of variables independent of each other.

On the other hand, the more the color of the tile tends toward *dark red*, the less there are observations represented by that tile compared to the expected quantity in the case of variables independent of each other.

Looking at the top row of tiles in *Figure 11.25* associated with *Survived = 0*, we can say that, among the non-survivors, more than 50% of the people belong to the third class and the number is larger than expected if there was no association between survivors and classes. So, in general, we can say that there is a *positive association* of medium strength between the non-survivors and the third-class people. In other words, it is quite likely to find third-class people among non-survivors. In contrast, the number of first-class people among non-survivors is far lower than would be expected if there were no association between survivors and classes. So there is a strong negative association between the non-survivors and the first-class people. Thus, it is very likely that first-class people are not present among non-survivors.

Focusing instead on the bottom row of tiles in *Figure 11.25*, we have confirmation of what we have already said: it is very likely to find first-class people among the survivors and very unlikely to find third-class people.

The numbers behind the plot in *Figure 11.25* are based on conditional probabilities and can be found by calculating the **contingency table** (also called **crosstab**) generated by the various elements of each of the categorical variables under analysis:

		Pclass			
Survived		1	2	3	Total
0	0	80	97	372	549
	1	133	113	303	549
	Total	14.6 %	17.7 %	67.8 %	100 %
1	0	37 %	52.7 %	75.8 %	61.6 %
	1	136	87	119	342
	Total	39.8 %	25.4 %	34.8 %	100 %
Total	0	63 %	47.3 %	24.2 %	38.4 %
	1	216	184	491	891
	Total	24.2 %	20.7 %	55.1 %	100 %
		100 %	100 %	100 %	100 %

$$\chi^2 = 102.889 \cdot df = 2 \cdot \boxed{\text{Cramer's } V = 0.340} \cdot p = 0.000$$

observed values

expected values

% within Survived

% within Pclass



Figure 11.26 – Mosaic plot of the variables Survived and Pclass

See the *Reference* section for more details about contingency tables and mosaic plots.

But in a nutshell, how do we numerically determine the global strength of the association between two categorical variables? The answer lies in the coefficient highlighted in *Figure 11.26*, namely Cramér's V. Let's see what this is all about.

Harald Cramér's correlation coefficient

Cramér's correlation coefficient (V) measures the strength of the association between two categorical variables and it ranges from 0 to +1 (it doesn't admit negative values, contrary to the coefficients seen until now). This coefficient is based on **Pearson's chi-square (χ^2) statistic**, which is used to test whether two categorical variables are independent. Cramér's V formula is as follows:

Figure 11.27 – Cramér's V formula

 Figure 11.27 – Cramér's V formula

In the formula of *Figure 11.27*, the value N is the sample size and k is the smallest number of distinct elements among those of each categorical variable in the association.

Cramér's V coefficient is a symmetric function and the guidelines of *Figure 11.28* can be used to determine the magnitude of its effect size:

 Figure 11.28 – Cramér's V effect size ranges
Figure 11.28 – Cramér's V effect size ranges

The fact that V is a symmetric function leads to an important loss of information. Let's see why.

Henri Theil's uncertainty coefficient

Suppose you have the two categorical variables `IsFraudster` and `Hobby` in the following dataset:

 Figure 11.29 – Sample dataset of categorical variables
Figure 11.29 – Sample dataset of categorical variables

Each value of the `Hobby` variable can be associated with a unique value of the `IsFraudster` variable (for example, *Chess* → *Fraudster*). However, the reverse is not true (for example, *Fraudster* → [*Chess*, *Body building*]). Therefore, the strength of the *Hobby* → *Fraudster* association (I know `Hobby` and need to determine `IsFraudster`) is of higher magnitude than the *Fraudster* → *Hobby* association (I know `IsFraudster` and need to determine `Hobby`). Unfortunately, the use of Cramér's coefficient V causes this distinction to be lost, since it is a symmetric function. To maintain the asymmetric nature of the relation, we must introduce Theil's uncertainty coefficient.

Theil's uncertainty coefficient, also called the **entropy coefficient**, between two variables, X and Y , has a range of $[0, 1]$ and it is defined as follows:

 Figure 11.30 – Theil's uncertainty coefficient formula
Figure 11.30 – Theil's uncertainty coefficient formula

It is based on the concept of **entropy**, which provides information about the variation or diversity (and therefore uncertainty) of the information contained in one variable, given by $H(X)$. Then it's also based on the **conditional entropy** (or **joint entropy**) $H(X|Y)$, which measures data diversity associated with the two variables X and Y .

For example, always considering the Titanic disaster dataset, the coefficient $U(\text{Survived}|\text{Pclass})$ is 0.06; conversely, $U(\text{Pclass}|\text{Survived})$ is 0.09.

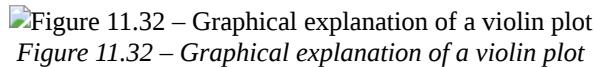
Let's see which coefficient to consider when dealing with associations in which one variable is numeric and the other is categorical.

Considering a numeric variable and a categorical one

If you want to graphically represent an association between a numeric variable and a categorical variable, the boxplot or the violin plot are the types of graphic representation for you. If you have already come across the problem of having to represent the distribution of a variable by highlighting key statistics, then you should be familiar with a **boxplot**:

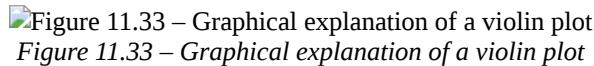
 Figure 11.31 – Graphical explanation of a boxplot
Figure 11.31 – Graphical explanation of a boxplot

A **violin plot** is nothing but a combination of a histogram/distribution plot and a boxplot for the same variable:



You will find more details about boxplots and violin plots in the references.

When you need to relate a numeric variable to a categorical one, you can build a violin plot for each element of the categorical variable. To go back to the Titanic disaster dataset example, taking the `Pclass` (categorical) and `Age` (numeric) variables into account, you get this multiple violin plot:



If you look carefully inside each violin plot, you will see a white dot for each thin black boxplot plotted within it. Those dots represent the mean of each distribution of the `Age` variable for each element of the `Pclass` variable. Since they are arranged at fairly distinct heights from each other, it is possible that the `Pclass` variable is a good predictor for the `Age` variable. But how do we measure the strength of the association between numeric and categorical variables? The answer is given to us by the *correlation ratio*.

Karl Pearson's correlation ratio

It's again thanks to Karl Pearson that we have a tool to calculate the degree of nonlinear association between a categorical and a numerical variable. This is the **correlation ratio** (η , eta), which was introduced during the study of analysis of variance (**ANOVA**), and it ranges from 0 to +1.

Just as r^2 can be interpreted as the percentage of variance in one variable explained linearly by the other, η^2 (also called the **intraclass correlation coefficient**) represents the percentage of variance in the dependent (target) variable explained *linearly or nonlinearly* by the independent (predictor) variable. For this interpretation to be valid, it requires that the dependent variable be numeric and the independent variable be categorical.

The correlation ratio is an asymmetric function only if the categorical variable is an ordinal one (that is, days of week can be transformed into integers), otherwise, it only makes sense in one way. The interclass correlation coefficient formula (from which we immediately derive the correlation ratio by applying the square root) is as follows:



The value is the mean of y broken down by category x and is the mean of the whole y (cross-category). Since σ represents the *variance* of a variable, we can read η^2 as the ratio of the dispersion of the variable y weighted for each category of x to the full dispersion of y .

Important Note

The more η^2 tends to 1, the less the observations disperse around their mean value for each individual category. Therefore, the total dispersion of the numerical variable is all due to the breakdown in categories, and not to the individual dispersions for each category. That is why in this case we can say that there is a strong association between the numerical variable and the categorical variable.

To better understand the concept laid out above, consider analyzing the grades taken in three subjects. If the grades are dispersed for each subject, you have a certain eta. If, on the other hand, the grades coincide for each subject, then eta equals 1. This statement is represented in *Figure 11.35*, where each observation is plotted as a point in the violin plots:

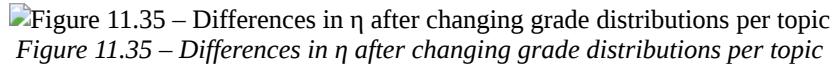


Figure 11.35 – Differences in η after changing grade distributions per topic
Figure 11.35 – Differences in η after changing grade distributions per topic

So, let's now implement the correlation coefficients described in this section in Python.

Implementing correlation coefficients in Python

The Python community is very lucky because Shaked Zychlinski developed a library with many data analysis tools, including a function that takes into account the data type of the columns of a pandas dataframe and generates a dataframe with the appropriate correlations. This library is **Dython** (<http://shakedzy.xyz/dython/>) and can be installed via `pip`. Moreover, if you want to create mosaic plots, you must also install **statsmodels** (<https://www.statsmodels.org/>) in your `pbi_powerquery_env` environment. As you've probably learned by now, proceed as follows:

1. Open the Anaconda prompt.
2. Enter the command `conda activate pbi_powerquery_env`.
3. Enter the command `pip install dython`.
4. Enter the command `pip install statsmodels`.

The code you'll find in this section is available in the file `03-titanic-disaster-analysis-in-python.py` in the `chapter11\Python` folder.

Of all the utilities in Dython, we need the ones in the `nominal` module. After loading the main libraries, we also create a helper function to draw a violin plot.

At this point, you can load the Titanic disaster data from a CSV file exposed on the web and transform the numerical columns `Survived` and `Pclass` (passenger class) into strings, since they are categorical variables:

```
dataset_url = 'http://bit.ly/titanic-data-csv'  
df = pd.read_csv(dataset_url)  
categ_cols = ['Survived', 'Pclass']  
df[categ_cols] = df[categ_cols].astype(str)
```

It is then possible to calculate *Cramér's V* coefficient between the above two categorical variables:

```
cramers_v(df['Survived'], df['Pclass'])
```

It returns a value of `0.34`, indicating a medium association strength.

You can also calculate *Theil's U* uncertainty coefficient of the `Survived` variable given the `Pclass` variable:

```
theils_u(df['Survived'], df['Pclass'])
```

The returned value is `0.087`. Conversely, you can calculate the same coefficient for the `Pclass` variable given the `Survived` one, in order to show the asymmetry of the function:

```
theils_u(df['Pclass'], df['Survived'])
```

This one returns `0.058`. So it's clear that the association `Pclass → Survived` is stronger than the opposite one.

How about calculating the *correlation ratio* η between the variables `Age` (passenger age) and `Pclass`? Let's do that:

```
correlation_ratio(categories=df['Pclass'], measurements=df['Age'])
```

The result is `0.366`. Now, what if you want to get a correlation value for each couple of columns regardless of their data type? In this case, the `associations()` function is our friend. You just have to specify whether you want to use Theil's U coefficient (`nom_nom_assoc = 'theil'`) or Cramér's V one (`nom_nom_assoc = 'cramer'`) for categorical variables and that's it:

```
ass = associations(df, nom_nom_assoc = 'theil',
                   num_num_assoc = 'pearson',
                   figsize=(10,10), clustering=True)
```

As a result, you'll get a beautiful heatmap that helps you understand at a glance which columns have the strongest correlation:

Figure 11.36 – Correlation heatmap
Figure 11.36 – Correlation heatmap

As you may have noticed, you can also select the type of correlation to be used between numeric variables (Pearson, Spearman, Kendall) via the `num_num_assoc` parameter. Moreover, you can access the coefficient dataframe using the code `ass['corr']`.

Let's now see how to implement the same things in R.

Implementing correlation coefficients in R

There is no R package on CRAN similar to Python that allows you to calculate correlations between columns in a tibble regardless of their data type. Nor is there a package in which the correlation ratio as previously defined is defined. Therefore, based on the source code of the Python package, we created them from scratch.

Instead, we used some CRAN packages that expose some of the correlation functions introduced in the previous sections. In particular, these packages are as follows:

- **rstatix** (<https://github.com/kassambara/rstatix>), an intuitive pipe-friendly framework for basic statistical tests. We used it for the `cramer_v()` function.
- **DescTools** (<https://andrisignorell.github.io/DescTools/>), a collection of miscellaneous basic statistics functions. We used it for the `UncertCoef()` function.
- **vcd** (<https://cran.r-project.org/web/packages/vcd/index.html>), a collection of visualization techniques and tools for categorical data.
- **sjPlot** (<https://strengejacke.github.io/sjPlot/>), a collection of plotting and table output functions for data visualization.

So, you need to install these packages in the most recent version of CRAN R you have:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('rstatix')`. Then press *Enter*.
3. Click on the **Console** window and enter this command: `install.packages('DescTools')`. Then press *Enter*.
4. Click on the **Console** window and enter this command: `install.packages('vcd')`. Then press *Enter*.
5. Click on the **Console** window and enter this command: `install.packages('sjPlot')`. Then press *Enter*.

The code you'll find in this section is available in the file `03-titanic-survive-class-analysis-in-r.R` in the `chapter11\R` folder.

After loading the most important libraries, again we created a helper function to graph a violin plot. After that, we defined the functions `correlation_ratio()` to calculate eta and `calc_corr()` to calculate the correlation of a tibble regardless of the data type of the given columns.

At this point, you can load the Titanic disaster data from a CSV file exposed on the web and transform the numerical columns `Survived` and `Pclass` (passenger class) into factors, since they are categorical variables:

```
dataset_url <- 'http://bit.ly/titanic-data-csv'
tbl <- read_csv(dataset_url)
tbl <- tbl %>%
  mutate( across(c('Survived', 'Pclass'), as.factor) )
```

It is then possible to calculate *Cramér's V* coefficient between the above two categorical variables:

```
rstatix::cramer_v(x=tbl$Survived, y=tbl$Pclass)
```

It returns a value of `0.34`, indicating a medium association strength.

You can also calculate *Theil's U* uncertainty coefficient of the `Survived` variable given the `Pclass` variable:

```
DescTools::UncertCoef(tbl$Survived, tbl$Pclass, direction = 'row')
```

The returned value is `0.087`. Also, in this case, we can calculate the *correlation ratio* η between the numeric variable `Age` and the categorical `Pclass` one:

```
correlation_ratio( categories = tbl$Pclass, measurements = tbl$Age, numeric_replace_value = 0)
```

The result is `0.366`. Now you can get the correlation value for each couple of columns regardless of their data type. We implemented the `calc_corr()` function for this purpose. You just have to specify whether you want to use Theil's U coefficient (`theil_uncert=True`) or Cramér's V one (`theil_uncert=False`) for categorical variables and that's it:

```
# Create two data frames having the only column containing the tibble column names as values
row <- data.frame(row=names(tbl))
col <- data.frame(col=names(tbl))
# Create the cross join data frame from the previous two ones
ass <- tidyverse::crossing(row, col)
# Add the corr column containing correlation values
corr_tbl <- ass %>%
  mutate(corr = map2_dbl(row, col, ~ calc_corr(data = tbl, row_name = .x, col_name = .y, theil_uncert=False)))
corr_tbl
```

You'll see something like this as a result:

Figure 11.37 – Correlation tibble you get from the Titanic one
Figure 11.37 – Correlation tibble you get from the Titanic one

You can also plot a heatmap of the correlation tibble just created using the following code:

```
corr_tbl %>%
  ggplot( aes(x=row, y=col, fill=corr) ) +
  geom_tile() +
  geom_text(aes(row, col, label = round(corr, 2)), color = "white", size = 4)
```

Here is the result:

Figure 11.38 – Heatmap of the correlation tibble
Figure 11.38 – Heatmap of the correlation tibble

Awesome! You have just implemented all of the correlation coefficients studied in this chapter in R as well. Let's now see how to implement them in Power BI, both with Python and with R.

Implementing correlation coefficients in Power BI with Python and R

Obviously, Power BI was not born as an advanced statistical analysis tool, so you will not find, for example, all you need to implement all the correlation coefficients seen in the previous sections. In this case, the support given by Python and R is of fundamental importance for this purpose.

Also, in this case, due to the simplicity of the code, we will implement the correlation coefficients in both Python and R in one project.

First, make sure that Power BI Desktop references the correct versions of Python and R in **Options**. Then follow these steps:

1. Click on **Get Data**, select **Web**, and click on **Connect**.
2. Enter the <http://bit.ly/titanic-dataset-csv> URL into the **URL** textbox and click **OK**.
3. You'll see a preview of the data. Then click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.
5. Enter the script you can find in the file `04-correlation-analysis-in-power-bi-with-python.py` into the `Chapter11\Python` folder.
6. We are only interested in the data in `result_df`. So, click its **Table** value.
7. You'll see the preview of all correlation coefficients between all the columns of the dataset.
8. Click **Home** on the ribbon and then click **Close & Apply**.
9. Repeat steps 1 to 3.
10. Click **Transform** on the ribbon and then **Run R script**.
11. Enter the script you can find in the file `04-correlation-analysis-in-power-bi-with-r.R` in the `Chapter11\R` folder.
12. We are only interested in the data in `corr_tb1`. So, click its **Table** value.
13. You'll see the preview of all correlation coefficients between all the columns of the dataset.
14. Click **Home** on the ribbon and then click **Close & Apply**.

Amazing! You have just calculated the correlation coefficients for all the numeric columns of a source dataset in Power BI with both Python and R!

Summary

In this chapter, you learned how to calculate the correlation coefficient for two numerical variables according to Pearson, Spearman, and Kendall. Then you also learned how to calculate it for two categorical variables thanks to Cramér's V and Theil's uncertainty coefficient. Finally, you also learned how to calculate it for one numeric and one categorical variable thanks to the correlation ratio.

In the next chapter, you will see how statistics are really important for determining outliers and imputing missing values in your dataset.

References

For additional reading, check out the following books and articles:

1. *Spearman's Rank-Order Correlation* (<https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php>)
2. *Concordant Pairs and Discordant Pairs* (<https://www.statisticshowto.com/concordant-pairs-discordant-pairs/>)
3. *Mosaic Plot and Chi-Square Test* (<https://towardsdatascience.com/mosaic-plot-and-chi-square-test-c41b1a527ce4>)
4. *Understanding Boxplots* (<https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51>)
5. *Violin plots explained* (<https://towardsdatascience.com/violin-plots-explained-fb1d115e023d>)
6. *The Search for Categorical Correlation* (<https://towardsdatascience.com/the-search-for-categorical-correlation-a1cf7f1888c9>)

12 Adding Statistics Insights: Outliers and Missing Values

In an effort to extend the data enrichment possibilities in Power BI through statistical functions, we will explore some methodologies to detect univariate and multivariate outliers in your dataset. In addition, advanced methodologies to impute possible missing values in datasets and time-series will be explored. Knowledge of these techniques is critical for the experienced analyst because Power BI does not provide useful tools for this purpose by default.

In this chapter, we will cover the following topics:

- What outliers are and how to deal with them
- Identifying outliers
- Implementing outlier detection algorithms
- What missing values are and how to deal with them
- Diagnosing missing values
- Implementing missing value imputation algorithms

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R With Power BI*, and *Chapter 3, Configuring Python with Power BI*.

What outliers are and how to deal with them

Generally, outliers are defined as those observations that lie at an *abnormal distance* from other observations in a data sample. In other words, they are *uncommon values* in a dataset. The abnormal distance we're talking about obviously doesn't have a fixed measurement but is strictly dependent on the dataset you're analyzing. Simply put, it will be the analyst who decides the distance beyond which to consider other abnormal distances based on their experience and functional knowledge of the business reality represented by the dataset.

Important Note

It makes sense to talk about outliers for numeric variables or for numeric variables grouped by elements of categorical variables. It makes no sense to talk about outliers for categorical variables only.

But why is there so much focus on managing outliers? The answer is that very often they cause undesirable macroscopic effects on some statistical operations. The most striking example is that of a linear correlation in the presence of an outlier in an "uncomfortable" position and the same calculated by eliminating the outlier:

Figure 12.1 – A simple scatterplot
Figure 12.1 – A simple scatterplot

As is evident in *Figure 12.1* and from what you learned from *Chapter 11, Adding Statistics Insights: Associations*, Pearson's correlation r suffers greatly from outliers.

But then, is it always sufficient to remove outliers *a priori* in order to solve any problems you might find downstream in your analysis? As you can imagine, the answer is "no," because it all depends on the type of outlier you are dealing with.

The causes of outliers

Before considering any action to be applied to the outliers of a variable, it is necessary to consider what might have generated them. Once the cause is established, it may be immediate to fix the outliers. Here is a possible categorization of the causes of outliers:

- **Data entry errors:** There may be an analyst collecting the data who made mistakes compiling it. If the analyst is collecting the birth dates of a group of people, it may be, for example, that instead of writing 1977, they write 177. If the dates that they have collected belong to the range from 1900 to 2100, it is quite easy to correct the outlier that has been created due to the entry error. Other times, it is not possible to recover the correct value.
- **Intentional outliers:** Very often, the introduction of "errors" is intentional by the individuals to whom the measurements apply. For example, adolescents typically do not accurately report the amount of alcohol they consume.
- **Data processing errors:** Data transformation processes that are usually applied to analytics solutions can introduce unintended errors, which in turn can give rise to possible outliers.
- **Sampling errors:** Sometimes, the data on which you perform your analysis must be sampled from a much larger dataset. It may be in this case that the analyst does not select a subset of data representing the entire population of data. For example, you need to measure the height of athletes, and, by mistake, you include some basketball players in your dataset.
- **Natural outliers:** So-called "natural" outliers exist because they are part of the nature of business and are not the result of any kind of error. For example, it's pretty much a given that shopping malls sell more products at Christmas time.

Once the nature of specific outliers is identified, it is certainly easier to try to correct them as much as possible. How do we proceed? There are a few common ways to correct outliers that can be considered.

Dealing with outliers

The most widely used approaches to deal with outliers are as follows:

- **Dropping them:** The analyst concludes that eliminating the outliers altogether guarantees better results in the final analysis.
- **Capping them:** It is common to use the strategy of assigning a fixed extreme value (cap) to all those observations that exceed it (in absolute value) when it is certain that all extreme observations behave in the same way as those with the cap value.
- **Assigning a new value:** In this case, outliers are eliminated by replacing them with null values, and these null values are imputed using one of the simplest techniques: the replacement of null values with a fixed value that could be, for example, the mean or median of the variable in question. You'll see more complex imputation strategies in the next sections.
- **Transforming the data:** When the analyst is dealing with natural outliers, very often the histogram of the variable's distribution takes on a skewed shape. Right-skewed distributions are very common, and if they were used as they appeared, many statistical tests that assume a normal distribution would give incorrect results. In this case, it is often used to transform the variable by applying a monotonic function, which in some way "straightens out" the imbalance (this is the case of the `log()` function, for example). Once transformed, the new variable satisfies the requirements of the tests and can therefore be analyzed without errors. Once the results have been obtained from the transformed variable, they must be transformed again by the inverse function of the one used at the beginning (if `log()` was used, then the inverse is `exp()`) in order to have values that are consistent with the business variable under analysis.

Now that you know the most common ways of dealing with outliers, you need to figure out how to identify them.

Identifying outliers

There are different methods used to detect outliers depending on whether you are analyzing one variable at a time (**univariate analysis**) or multiple variables at once (**multivariate analysis**). In the univariate case, the analysis is

fairly straightforward. The multivariate case, however, is more complex. Let's examine them in detail.

Univariate outliers

One of the most direct and widely used ways to identify outliers for a single variable is to make use of boxplots, which you learned about in *Chapter 11, Adding Statistics Insights: Associations*. Some of the key points of a boxplot are the **interquartile range (IQR)**, defined as the distance from the **first quartile (Q1)** to the **third quartile (Q3)**, the **lower whisker** ($Q1 - 1.5 \times IQR$), and the **upper whisker** ($Q3 + 1.5 \times IQR$):

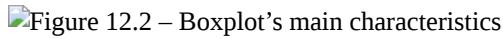


Figure 12.2 – Boxplot's main characteristics
Figure 12.2 – Boxplot's main characteristics

Specifically, all observations that are before the lower whisker and after the upper whisker are identified as outliers. This one is also known as **Tukey's method**.

Identification becomes more complicated when dealing with more than one variable.

Multivariate outliers

Identifying outliers when you are handling more than one variable (**multivariate outliers**) is not always straightforward. It depends on the number of variables in play and their data type.

Numeric variable and categorical variable

As long as you need to analyze how a numerical variable is distributed among the different elements of a categorical variable, it is still feasible with the tools seen so far. In fact, just plot a boxplot for the values of the numeric variable grouped by each element of the categorical variable:

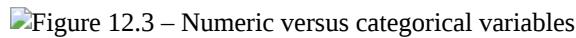


Figure 12.3 – Numeric versus categorical variables
Figure 12.3 – Numeric versus categorical variables

In fact, it may be that the numerical variable does not present any outliers but reveals some when it is broken down by the elements of the categorical variable.

All numeric variables

Generally, the inexperienced analyst tends to simplify the determination of outliers in multidimensional cases when there are only numeric variables.

Important Note

One might assume that an observation that is extreme in any variable is also a multivariate outlier, and this is often true. However, the opposite is not true: when variables are correlated, one can have a multivariate outlier that is not a univariate outlier in any variable.

When dealing with only numeric variables, it is still possible to use algorithms that measure the distance from the center of the distribution. Let's take the case of two numerical variables, which allows us to visualize the outliers using a scatterplot:

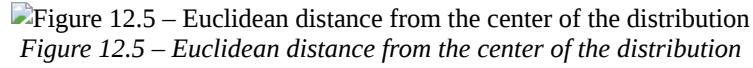


Figure 12.4 – Scatterplot of two numeric variables
Figure 12.4 – Scatterplot of two numeric variables

As you can see in *Figure 12.4*, we have also added in the margin the two boxplots for every single variable under analysis to verify that for each of them there are no outliers, except for the one detected at the bottom. You can also

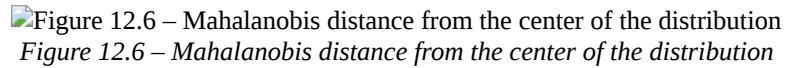
see that there is one outlier that is clearly different from all other observations but is not detected as an outlier by the two boxplots.

Imagine fixing a hypothetical center of the distribution and defining a distance from the center (Euclidean distance) above which the observations are to be considered outliers:

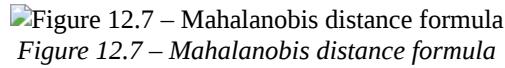
Figure 12.5 – Euclidean distance from the center of the distribution
Figure 12.5 – Euclidean distance from the center of the distribution

The above rule defines a circle centered at the center of the distribution. Considering a circle with the radius you see in *Figure 12.5*, you are going to identify several outliers (perhaps false positives?) that were not identified by looking at the boxplots alone, but the outlier you failed to identify before remains unidentified in this case as well. As you can well understand, the problem is that the distribution has an ellipsoidal shape that is distributed along the main diagonal of the Cartesian plane. Using a circle is certainly ill-suited to fit a distribution of a different shape.

What if there was a distance that also considered the shape of the distribution? This is precisely the case with the **Mahalanobis distance**. This new distance differs from the others because it considers the **covariance** between the two variables. Covariance and Pearson's correlation are two quantities associated with very similar concepts, so in some cases they are interchangeable (take a look at the references). The fact that the Mahalanobis distance accounts for the correlation between the two variables is evident in *Figure 12.6*:

Figure 12.6 – Mahalanobis distance from the center of the distribution
Figure 12.6 – Mahalanobis distance from the center of the distribution

For the curious, this is the formula to calculate it:

Figure 12.7 – Mahalanobis distance formula
Figure 12.7 – Mahalanobis distance formula

is a multivariate observation, \bar{x} is the multivariate mean of all observations, and S is the covariance matrix. The fact that the Mahalanobis distance depends on the mean of all observations (a very unstable measure, very sensitive to outliers) and the covariance matrix makes you realize that the same limitations encountered for the Pearson's coefficient apply, which are as follows:

- Possible outliers at inconvenient locations could greatly affect the multivariate center defined by the mean of all observations. If the center is not well calculated, it is very likely that the resulting application of the Mahalanobis distance will identify erroneous outliers. This is easily solved by computing the center via a median-based formula, which is much more robust to the presence of extreme observations.
- If extreme outliers are present, the covariance matrix may also be negatively affected. This problem is also solved by adopting a robust version of the covariance matrix using the **Minimum Covariance Determinant (MCD)**. This method, in addition to providing a robust covariance matrix, also returns a robust estimate of the center of the observations.
- It is very likely that the usage of the Mahalanobis distance will return false outliers in case of *skewed*, *nonlinear*, or *heteroscedastic* distributions. These are the cases in which it is necessary to resort to transformations of the variables involved, as far as possible, before applying distance calculations. The goal of the transformations is to obtain distributions that are as normal as possible and to make the associations between the various variables as linear as possible. In these cases, **Box-Cox transformations** or **Yeo-Johnson transformations** are used.

The identification of outliers becomes much more complicated when dealing with mixed variables (numerical and categorical) in numbers greater than two. It is necessary to use different data science techniques (feature engineering techniques for categorical variables, handling of unbalanced datasets, and so on) and to apply specific machine learning anomaly detection algorithms. For this reason, cases of this type are out of scope.

Once the outliers (both univariate and multivariate) are identified, it is up to the analyst to decide which method to adopt to try to fix it, if possible.

Let's now see how to implement outlier detection algorithms according to what you learned in the previous sections.

Implementing outlier detection algorithms

The first thing you'll do is implement what you've just studied in Python.

Implementing outlier detection in Python

In this section, we will use the *Wine Quality* dataset created by Paulo Cortez et al. (<https://archive.ics.uci.edu/ml/datasets/wine+quality>) to show how to detect outliers in Python. The dataset contains as many observations as the different types of red wine, each described by the organoleptic properties measured by the variables, except for the `quality` one, which provides a measure of the quality of the product using a discrete grade scale from 1 to 10.

You'll find the code used in this section in the `01-detect-outliers-in-python.py` file into the `Chapter12\Python` folder.

Once you have loaded the data from the `winequality-red.csv` file directly from the web into the `df` variable, let's start by examining the `sulphates` variable. Let's check if it contains any outliers by displaying its boxplot, which was obtained through a wrapper function that we have defined in the code:

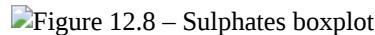
Figure 12.8 – Sulphates boxplot

Figure 12.8 – Sulphates boxplot

Apparently there are plenty of values after 1.0. To be able to locate them in the dataset, we created a function that accepts a dataframe as input and the name of the numeric column to be considered, and as output returns the dataframe with the addition of a column of Boolean values, containing `True` when the value of the column is an outlier, and `False` otherwise:

```
def add_is_outlier_IQR(data, col_name):
    col_values = data[col_name]

    Q1=col_values.quantile(0.25)
    Q3=col_values.quantile(0.75)
    IQR=Q3-Q1

    outliers_col_name = f'is_{col_name.replace(" ", "_")}_outlier'
    data[outliers_col_name] = ((col_values < (Q1 - 1.5 * IQR)) | (col_values > (Q3 + 1.5 * IQR)))

    return data
add_is_outlier_IQR(df, 'sulphates')
```

Once we have identified the outliers of the initial distribution of the `sulphates` variable, we can draw its boxplot by removing the outliers to see what changes:

```
df_no_outliers = df.loc[~df['is_sulphates_outlier']]
```

The resulting boxplot is as follows:

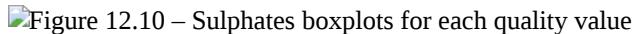
Figure 12.9 – Sulphates boxplot once outliers were removed

Figure 12.9 – Sulphates boxplot once outliers were removed

As you can see, some outliers are still visible. This is due to the fact that removing the outliers from the initial distribution caused the distribution to change (its statistical properties changed). So, what you see in *Figure 12.9*

are the outliers of the new distribution that was created.

As already explained, it is up to the analyst to figure out if the outliers identified can be corrected in some way, eliminated, or left where they are. Suppose in this case that the outliers in the second distribution are natural outliers. Let's try to break the new distribution down to the individual values of the `quality` variable and draw a boxplot for each of them:

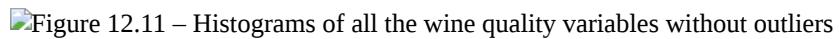

Figure 12.10 – Sulphates boxplots for each quality value

As shown in *Figure 12.10*, the distribution of sulphates for wines that received a grade of 5 has several outliers. This could induce the analyst to try to understand how much the presence of sulphates affects the final rating given by users to the wine, paying particular attention to the case of a wine considered of average quality.

If, on the other hand, we wanted to identify multivariate outliers for all numerical variables in the dataset, excluding the `quality` variable, we need to change our approach by trying to apply the Mahalanobis distance, as you learned in the previous section. We assume that the elimination of outliers for each individual variable has been validated. So, let's now try to figure out if multivariate outliers are present for the numeric variables in the `df_no_outliers`' dataframe. First, however, it is necessary to check whether the distributions of the variables under analysis are skewed. Therefore, we try to draw a histogram for each of the variables:

```
df_no_outliers.drop('quality', axis=1).hist(figsize=(10,10))
plt.tight_layout()
plt.show()
```

The resulting plot is as follows:


Figure 12.11 – Histograms of all the wine quality variables without outliers

It is evident that some of them are extremely right-skewed (*residual sugar, chlorides, total sulfur dioxide*, etc.), therefore it is necessary to try to apply transformations that attempt to “normalize” the single distributions. Generally, *Box-Cox transformations* are applied. But since in this case some values of the distributions are not positive, it is not possible to apply them. It is therefore necessary to use other transformations that have the same objective, named *Yeo-Johnson*. For more details on these transformations, check out the references.

For convenience, we created a wrapper function that transforms a pandas dataframe of only numeric variables by applying Yeo-Johnson transformations and also returns the corresponding lambda values:

```
from sklearn.preprocessing import PowerTransformer
def yeo_johnson_transf(data):
    pt = PowerTransformer(method='yeo-johnson', standardize=True)
    pt.fit(data)
    lambdas = pt.lambdas_
    df_yeojohnson = pd.DataFrame( pt.transform(data), columns=data.columns.values )
    return df_yeojohnson, lambdas
```

Then, once you've transformed the dataframe into an object, you can try drawing histograms of the distributions of the transformed variables to see if the skewness has been smoothed out:

```
df_transf, lambda_arr = yeo_johnson_transf(df_no_outliers[numerical_col_names])
df_transf.hist(figsize=(10,10))
plt.tight_layout()
plt.show()
```

This is the plot you get:


Figure 12.12 – Histograms of all the wine quality variables transformed

Figure 12.12 – Histograms of all the wine quality variables transformed

It is quite evident that now the distributions look more like the "bells" of normal distributions. You can now calculate Mahalanobis distances with the surety that it will detect outliers with fewer errors.

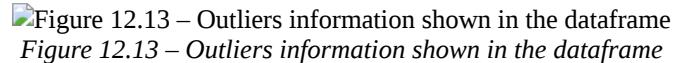
The identification of outliers is done using a robust estimator of covariance, the *MCD*. Since the squared Mahalanobis distance behaves as a Chi-Squared distribution (see the references), we can calculate the threshold value above which to consider an observation an outlier thanks to this distribution, passing the desired cutoff value to its *percent point function* (*ppf()*):

```
from sklearn.covariance import MinCovDet
robust_cov = MinCovDet(support_fraction=0.7).fit(df_transf)
center = robust_cov.location_
D = robust_cov.mahalanobis(df_transf - center)
cutoff = 0.98
degrees_of_freedom = df_transf.shape[1]
cut = chi2.ppf(cutoff, degrees_of_freedom)
```

Once you have determined the threshold value, you can add two columns to the dataframe: a column that identifies whether the observation (the row) is an outlier according to Mahalanobis and a column that reports the probability that an observation is not an outlier by chance:

```
is_outlier_arr = (D > cut)
outliers_stat_proba = np.zeros(len(is_outlier_arr))
for i in range(len(is_outlier_arr)):
    outliers_stat_proba[i] = chi2.cdf(D[i], degrees_of_freedom)
df['is_mahalanobis_outlier'] = is_outlier_arr
df['mahalanobis_outlier_stat_sign'] = outliers_stat_proba
df[df['is_mahalanobis_outlier']]
```

You'll see a dataframe chunk like this:


Figure 12.13 – Outliers information shown in the dataframe

Wow! With a minimum of statistics knowledge, you were able to identify multivariate outliers of numeric variables in Python.

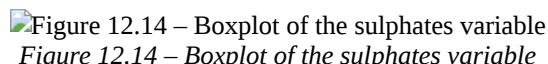
It is possible to get the same results with R. Let's see how.

Implementing outlier detection in R

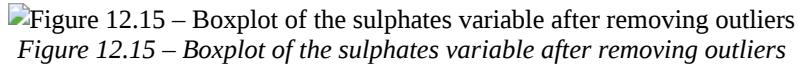
You'll find the code used in this section in the `01-detect-outliers-in-r.R` file in the `Chapter12\R` folder. In order to run it properly, you need to install new packages:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('robust')`. Then press *Enter*.
3. Enter this command: `install.packages('recipes')`. Then press *Enter*.

Once you have loaded the data from the `winequality-red.csv` file directly from the web into the `df` variable, you'll draw the boxplot of the `sulphates` variable using the `boxPlot()` wrapper function:


Figure 12.14 – Boxplot of the sulphates variable

Since there are many outliers visible in *Figure 12.14*, they are identified using the `add_is_outlier_IQR()` function, which adds an identifier column to the dataframe. As the name indicates, the function determines the outliers based on the interquartile range. At this point, the boxplot of the same variable is drawn again, this time eliminating the previously identified outliers:



Assuming you now want to identify multivariate outliers, it is worthwhile to first look at the histograms of the individual variables to see if significant skewness is present. The histograms are drawn using the following `dataframeHist()` function:

```
dataframeHist <- function(data, bins = 10) {
  data %>%
    tidyr::pivot_longer( cols = everything() ) %>%
    ggplot( aes(value) ) +
    geom_histogram( fill='orange', na.rm = TRUE, bins = bins ) +
    theme( ... ) +
    facet_wrap(~ name, scales = "free")
}
```

A special feature of this function is the use of the `pivot_longer()` function of the `tidyverse` package on all columns to verticalize their names in the new `name` column, to which correspond the initial values in the new `value` column. The result is as follows:



Since the skewness is obvious, you can apply Yeo-Johnson transformations thanks to the `yeo_johnson_transf()` wrapper function we created for you. The peculiarity of this function is that it makes use of a ready-made step in the `recipes` package, which facilitates the whole pre-processing phase. To learn more about the use of `recipes`, take a look at the references.

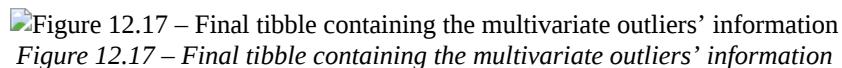
As you learned in the previous section, the Yeo-Johnson transformations solve the skewness problem quite well in this case. Therefore, it is possible to try applying Mahalanobis distance to detect outliers via the following code:

```
data <- df_transf %>%
  select( numeric_col_names )
cov_obj <- data %>%
  covRob( estim="mcd", alpha=0.7 )
center <- cov_obj$center
cov <- cov_obj$cov
distances <- data %>%
  mahalanobis( center=center, cov=cov )
```

At this point, given a cutoff value associated with the statistical significance with which we want to determine outliers, you can obtain the corresponding threshold value above which to consider an observation an outlier. Once the threshold is calculated, it is trivial to create an indicator column for the outliers. It is also possible to add a column indicating the probability with which an observation can be considered an outlier not by chance thanks to the `pchisq()` function:

```
cutoff <- 0.98
degrees_of_freedom <- ncol(data)
outliers_value_cutoff <- qchisq(cutoff, degrees_of_freedom)
data <- data %>%
  mutate(
    is_mahalanobis_outlier    = distances > outliers_value_cutoff,
    mahalanobis_outlier_proba = pchisq(distances, ncol(data)) )
data %>% filter( is_mahalanobis_outlier == TRUE )
```

The final result partially presents this output:



Way to go! You were able to identify multivariate outliers in R as well.

At this point, it is trivial to apply the Python and R code seen so far to Power BI.

Implementing outlier detection in Power BI

Power BI has tools that allow you to view outliers graphically and then analyze them by hovering. One of these was introduced in the November 2020 release and is the **Anomaly Detection** feature. The other one is the **Outliers Detection** custom visual. Let's see what the main differences are:

- **Anomaly Detection** is available directly in Power BI once you enable it as a preview feature (at the moment, it is in preview). It is *only supported for line chart visuals* containing *time-series data* in the Axis field.
- **Outliers Detection** is an open source R custom visual, and it has to be installed separately (<https://bit.ly/power-bi-outliers-detection>). There are five different implemented methods to detect outliers, and it works well for univariate and bivariate datasets. Multivariate datasets are to be avoided.

As you may have noticed, both of these tools are Power BI visuals.

Important Note

All transformations performed within Python or R visuals modify the dataframe that will later be the object of visualization, but the changes cannot be persisted in the data model in any way.

It is precisely for this reason that we have decided to illustrate some methods for detecting outliers that can be applied in Power Query using Python or R. This way, you can identify observations that are outliers by simply filtering your data model tables appropriately. Due to the simplicity of the code, in this case we will also implement the correlation coefficients in both Python and R in one project.

First, make sure that Power BI Desktop references the correct versions of Python and R in the **Options**. Then follow these steps:

1. Click on **Get Data**, search for **web**, select **Web**, and click on **Connect**.
2. Enter <https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv> into the URL textbox and click **OK**.
3. You'll see a preview of the data. Then click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.
5. Enter the script you can find in the `02-detect-outliers-in-power-bi-with-python copy.py` file in the `Chapter12\Python` folder.
6. We are only interested in the data in `dataset`. So, click on its **Table** value.
7. You'll see a preview of the dataset that also has the new columns to identify outliers.
8. Click **Home** on the ribbon and then click **Close & Apply**.
9. Repeat steps 1 to 3.
10. Click **Transform** on the ribbon and then **Run R script**.
11. Enter the script you can find in the `02-detect-outliers-in-power-bi-with-r.R` file in the `Chapter12\R` folder.
12. We are only interested in the `data` tibble. So, click on its **Table** value.
13. You'll see the preview of the dataset that also has the new columns to identify outliers.
14. Click **Home** on the ribbon and then click **Close & Apply**.

Amazing! You have just identified outliers of a numeric dataset in Power BI with both Python and R!

What missing values are and how to deal with them

Data describing real-world phenomena often has a lot of missing data. Lack of data is a fact that cannot be overlooked, especially if the analyst wants to do an advanced study of the dataset to understand how much the variables in it are correlated.

The consequences of mishandling missing values can be many:

- The *statistical power* of variables with missing values is diminished, especially when a substantial number of values is missing for a single variable.
- The *representativeness of the dataset* subject to missing values may also be diminished, and thus the dataset in question may not correctly represent the substantive characteristics of the set of all observations of a phenomenon.
- Any statistical estimates may not converge to whole population values, thus *generating bias*.
- The results of the analysis conducted may not be correct.

But let's see what the causes could be that generate missing values in a dataset.

The causes of missing values

There can be many causes for a lack of values, determined by intentional or unintentional behaviors. Here is a non-exhaustive list:

- Corruption of data due to errors in writing, reading, or transmitting it
- Replacing outliers that excessively skew the dataset with null values
- Refusing to answer a question on a survey
- Lack of knowledge of the issues asked in a survey question

However, all of these causes can be summarized into *four types of cases*:

- **Missing Completely at Random (MCAR)**: The causes that generate the null values are totally independent both of the hypothetical values they would have if valorized (Y) and of the values of the other variables in the dataset (X). They just depend on external variables (Z). The advantage of data that is MCAR from a statistical perspective is that the dataset consisting of only complete values for both variables X and Y is an *unbiased sample* of the entire population. Unfortunately, MCAR cases are rarely found in real-world data.
- **Missing At Random (MAR)**: The missing data of a partially incomplete variable (Y) is related to some other variables in the dataset that do not have null values (X), but not to the values of the incomplete variable itself (Y). The dataset consisting of only complete values for both variables X and Y in the MAR case constitutes a *biased sample* of the entire population because it will surely miss all those values of X on which the null values of Y depend, resulting in a dataset that is not representative of the entire phenomenon. MAR is a more realistic assumption than MCAR.
- **Missing Not at Random due to external variables Z (MNAR Z)**: The missing data of a partially incomplete variable (Y) depends on variables not included in the dataset (external variables). For example, given a dataset without the variable “sex,” there may be observations for which the age value is zero. It could be possible that the respondents not providing this information are mostly women, since they stereotypically do not want to reveal their age. Therefore, eliminating observations that have non-null values for the “age” variable would generate a *biased dataset*.
- **Missing Not at Random due to missing values Y (MNAR Y)**: The missing data of a partially incomplete variable (Y) depends on the hypothetical values they would have if valorized. For example, it is well known that adolescents tend to never disclose the fact that they consume alcohol. Therefore, if we remove from the dataset those observations for which the value for alcohol consumption is null, implicitly we risk removing from the dataset most of the observations pertaining to adolescents, thus obtaining a *biased dataset*.

There are also statistical tests that allow you to understand if the distribution of missing data is MCAR (look at the references). But, as already mentioned, cases of MCAR are so rare that it is better to assume that the distribution of missing values of a dataset under consideration is either MAR or MNAR.

Depending on the type of missing data distribution, specific strategies can be employed to sanitize the missing values.

Handling missing values

The first thing to do, when possible, is to understand together with the referent of the data the reason for the missing values in the dataset, and whether it is possible to recover them. Unfortunately, most of the time it is not possible to recover missing data from the source and therefore different strategies must be adopted, depending on the case.

Easy imputation by hand

There may be cases of variables that are *obvious to impute by hand*. For example, in correspondence of the “blue” value of the variable “color” you will notice that the variable “weight” always takes the value of 2.4, except in a few cases where it is null. In those cases, it is easy to impute the missing values of the variable “weight” in relation to the “blue” color with the value 2.4.

Discarding data

The first solution to the missing values that comes to the analyst's mind is surely to eliminate the problem at the source, that is, to eliminate missing values. There are several ways to eliminate them:

- **Listwise or Complete-Case Analysis (CCA) deletion:** This method involves *deleting any observation (row) that contains at least one missing data element in any variable*. It is often applied when the number of missing values is low, and the number of observations is sufficiently large. As you have seen in the classification of the four types of missing data, the only case in which adopting this solution doesn't result in a biased dataset is the MCAR, a very rare case among datasets describing real-world phenomena. Listwise deletion is therefore not a good strategy when you are not faced with a case of MCAR with a sufficiently high number of observations in the dataset.
- **Pairwise or Available-Case Analysis (ACA) deletion:** Depending on the variables considered in a statistical analysis, this method *eliminates only those observations (rows) that have null values for the only variables involved*. Null values present in variables that are not involved in the analysis are not a reason to eliminate observations. Again, adopting this method does not generate a biased dataset only if the case under analysis is MCAR. The most obvious disadvantage of this method is that if you need to compare different analyses, you cannot apply it because the number of observations in the sample varies as the variables involved in the different analyses vary.
- **Variable deletion:** This method considers *removing the entire variable from the analysis* under study (and not from the dataset a priori!) when the proportion of missing values ranges is 60% and above. It makes sense to eliminate a variable if, after careful study, it is concluded that it does not contain important information for the analysis at hand. Otherwise, it is always preferable to try a method of imputation. Generally, the elimination of a variable is always the last option and should only be considered if the final analysis actually benefits from it.

When the analyst still has to heal the problem of missing values, even after trying to apply these elimination techniques, they must then resort to imputation techniques. Let's look at the most commonly used methods.

Mean, median, and mode imputation

This is an intuitively attractive method, also known as **single imputation**, for which you fill missing values with predefined values. Simplicity is unfortunately countered by some not negligible issues.

Perhaps the most common substitution of null values is with the **mean** value of the variable's distribution resulting from ignoring missing values. The motivation behind this choice is that *the mean is a reasonable estimate of an observation drawn at random from a normal distribution*. However, if the distribution in question is skewed, the analyst runs the risk of making severely biased estimates even if the dataset's missing value distribution is MCAR.

The skewness problem can be solved by using the **median** of the variable. However, the fact remains that the common problem in single imputation is replacing a missing value with a single value and then treating it as if it were a true value. As a result, single imputation ignores uncertainty and almost always underestimates variance

(remember that variance is synonymous with information; a variable with 0 variance is a constant value variable that usually does not enrich statistical analyses).

Mode (the value that is repeated most often) imputation is often used with categorical data represented as numbers. Even this method, when used without having strong theoretical grounds, introduces bias, so much so that sometimes analysts prefer to create a new category specifically for missing values.

Multiple imputation is often preferable to single imputation as it overcomes the problems of underestimating variance by considering both within-imputation and between-imputation variance. Let's see what this is all about.

Multiple imputation

It is thanks to Donald B. Rubin that in 1987 a methodology to deal with the problem of underestimation of variance in the case of single imputation was made public. This methodology goes by the name of **multiple imputation** and consists of the following steps:

1. **Imputation:** This step is very similar to the single imputation step, except that this time, values are extracted m times from a distribution for each missing value. The result of this operation is a set of m imputed datasets, for which all observed values are always the same, with different imputed values depending on the uncertainty of the respective distributions.
2. **Analysis:** You use all m imputed datasets for the statistical analysis you need to do. The result of this step is a set of m results (or analyses) obtained by applying the analysis in question to each of the m imputed datasets.
3. **Pooling:** The m results are combined in order to obtain unbiased estimates with the correct statistical properties. The m estimates of each missing value are pooled in order to have an estimated variance that combines the usual sampling variance (**within-imputation variance**) and the extra variance caused by missing data (**between-imputation variance**).

The whole process can be summarized by *Figure 12.18*:



Figure 12.18 – Multiple imputation process
Figure 12.18 – Multiple imputation process

Multiple imputation can be used in cases where the data is MCAR, MAR, and even when the data is MNAR if there are enough auxiliary variables.

The most common implementations of multiple imputation are as follows:

- **Multivariate Imputation by Chained Equations (MICE):** This imputes missing values focusing on one variable at a time. Once the focus is on one variable, MICE uses all other variables in the dataset (or an appropriately chosen subset of those variables) to predict missing values in that variable. Predictions of missing values are based on linear regression models for numerical variables and logistic regression models for categorical variables.
- **Amelia II:** This is named after Amelia Mary Earhart, an American aviation pioneer who, during an attempt to become the first woman to complete a global circumnavigation flight in 1937, disappeared over the central Pacific Ocean. Amelia II combines a bootstrapping-based algorithm and an **Expectation–Maximization (EM)** algorithm, making it fast and reliable. It also works very well for time-series data.

Recently, multiple imputation has been implemented using deep learning algorithms as well. In particular, the **Multiple Imputation with Denoising Autoencoders (MIDAS)** algorithm offers significant advantages in terms of accuracy and efficiency over other multiple imputation strategies, particularly when applied to large datasets with complex features.

Univariate time-series imputation

The problem of missing data afflicts not only multivariate tabular datasets, but also time-series datasets. For example, sensors that constantly collect data about a phenomenon could stop working at any time, generating holes

in the series. Often, the analyst is faced with a time-series that has missing values and must somehow impute these values because the processes to which the series do not handle null values.

The constraint of the consequentiality of events given by the temporal dimension forces the analyst to use specific imputation methods for time-series. Let's look at the most commonly used methods:

- **Last Observation Carried Forward (LOCF), Next Observation Carried Backward (NOCB):** In the LOCF method, the last observed (that is, non-null) measure of the variable in question is used for all subsequent missing values. The only condition in which LOCF is unbiased is when the missing data is completely random, and the data used as the basis for LOCF imputation has exactly the same distribution as the unknown missing data. Since it can never be proven that these distributions are exactly the same, all analyses that make use of LOCF are suspect and will almost certainly generate biased results. The NOCB method is a similar approach to LOCF, but works in the opposite direction, taking the first (non-null) observation after the missing value and replacing it with the missing value. It obviously has the same limitations as LOCF.
- **Exponentially Weighted Moving Average (EWMA):** In general, the moving average is commonly used in time-series to smooth out fluctuations due to short-term effects and to highlight long-term trends or cycles. EWMA is designed such that older observations are given lower weights. The weights decrease exponentially as the observation gets older (hence the name “exponentially weighted”). Missing values are imputed using the values of the resulting “smoothed” time-series.
- **Interpolation:** The interpolation technique is one of the most widely used techniques to impute missing data from a time-series. The basic idea is to use a simple function (such as a linear function, a polynomial function, or a spline function) that fits with the non-zero points near the missing value, then interpolate the value for the missing observation.
- **Seasonally Decomposed Imputation:** If the time-series under analysis has seasonality, this method could give very good results. The procedure adopted is to remove the seasonal component from the time-series, perform the imputation on the seasonally adjusted series, and then add the seasonal component back.

There are also algorithms to impute missing values for multivariate time-series.

Multivariate time-series imputation

This topic is beyond the scope of this chapter, but we simply wanted to specify that the *Amelia II* algorithm we discussed earlier is also used to impute missing values in a multivariate time-series, whereas it is not suitable for imputation on univariate time-series.

In order to figure out whether to impute missing values, we must first identify them in the dataset. Let's see how to do that.

Diagnosing missing values in R and Python

Before thinking about imputing missing values in a dataset, we must first know the extent to which the missing values affect each individual variable.

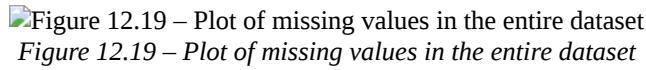
You can find the code used in this section in the `Chapter12\R\03-diagnose-missing-values-in-r.R` and `Chapter12\Python\03-diagnose-missing-values-in-python.py` files. In order to properly run the code and the code of the following sections, you need to install the requisite R and Python packages as follows:

1. Open the Anaconda prompt.
2. Enter the `conda activate pbi_powerquery_env` command.
3. Enter the `pip install missingno` command.
4. Enter the `pip install upsetplot` command.
5. Then, open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
6. Click on the **Console** window and enter `install.packages('naniar')`. Then press *Enter*.
7. Enter `install.packages('imputeTS')`. Then press *Enter*.
8. Enter `install.packages('forecast')`. Then press *Enter*.

9. Enter `install.packages('ggpubr')`. Then press *Enter*.
10. Enter `install.packages('missForest')`. Then press *Enter*.
11. Enter `install.packages('mice')`. Then press *Enter*.
12. Enter `install.packages('miceadds')`. Then press *Enter*.

Let's see at this point what features will come in handy when you face the analysis of missing values in a dataset.

The R package `naniar` provides the `vis_miss()` function, which displays in a single image the missing values of the whole dataframe:



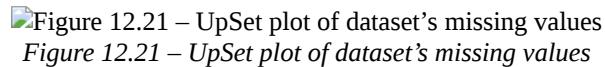
You can draw similar graphs in Python thanks to the `missingno` library (<https://github.com/ResidentMario/missingno>).

Knowing only the percentage value of the number of missing values compared to the total number of values of the variable under consideration can be limiting. That's why it's often useful to also know the details for each column via the `miss_var_summary()` function:



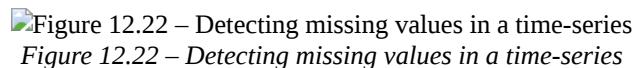
We developed a similar function in Python in the code you can find in the repository.

It would be interesting to be able to visualize combinations of missing values and missing intersections between variables. The R package `naniar` (<https://github.com/njtierney/naniar>) allows you to do just this kind of analysis thanks to the `gg_miss_upset()` function:

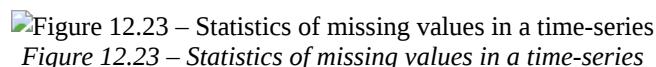


To achieve the same plot in Python, the process is a bit more complicated. You must first use the `upsetplot` module (<https://github.com/jnothman/UpSetPlot>). The problem lies in providing the `upSet()` function exposed by this package with an input dataframe in the required format. For this reason, we made the helper function `upsetplot_miss()` that you will find in the code to easily create the upset plot of missing values in Python as well.

In case you need to get an idea of the missing values in a time-series, the `imputeTS` R package provides the `ggplot_na_distribution()` function, which shows very clearly the holes in the time-series:



If, on the other hand, you need to get more complete details about the statistics of missing values in a time-series, the `statsNA()` function is for you:



Once you have carefully studied the distributions of the missing values of each variable and their intersections, you can decide which variables to leave in the dataset and which to submit to the various imputation strategies. Let's see how to do imputation in R and Python.

Implementing missing value imputation algorithms

From here on, all missing value analysis will be done in R because very statistically specialized and simple-to-use packages that do not exist in the Python ecosystem have been developed for this language.

Suppose we need to calculate the Pearson correlation coefficient between the two numerical variables, `Age` and `Fare`, of the Titanic disaster dataset. Let's first consider the case where missing values are eliminated.

Removing missing values

The impact of applying listwise and pairwise deletion techniques is evident in the calculation of Pearson's correlation between numerical variables in the Titanic dataset. Let's load the data and select only numeric features:

```
library(dplyr)
dataset_url <- 'http://bit.ly/titanic-data-csv'
tbl <- readr::read_csv(dataset_url)
tbl_num <- tbl %>%
  select( where(is.numeric) )
```

If you now calculate the correlation matrix for the two techniques separately, you will notice the differences:

```
# Listwise deletion
cor( tbl_num, method = 'pearson', use = 'complete.obs' )
# Pairwise deletion
cor( tbl_num, method = 'pearson', use = 'pairwise.complete.obs' )
```

You will see the following result:

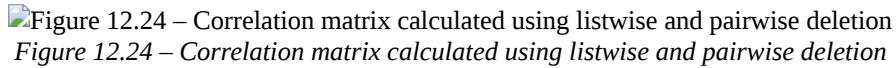


Figure 12.24 – Correlation matrix calculated using listwise and pairwise deletion
Figure 12.24 – Correlation matrix calculated using listwise and pairwise deletion

Let's see how to impute missing values in the case of a tabular dataset.

Imputing tabular data

You can find the code used in this section in the `Chapter12\R\04-handle-tabular-missing-values-in-r.R` file.

Again, starting with the Titanic disaster dataset, the first thing you need to do is remove the `Name` and `Ticket` columns because they have a high number of distinct values.

Important Note

It is important to eliminate categorical variables that have a high number of distinct values because otherwise, the MICE algorithm would fail due to the excessive RAM required. Generally, variables with high cardinality are not useful for the imputation of null values of other variables. There are cases in which the information contained in these variables could be fundamental for the imputation (for example, zip codes). In this case, it is necessary to use transformations that reduce the cardinality without losing the information contained in the variables. For further information, take a look at the references.

Since the missing values in the `Cabin` column represent more than 70% of all values, we decided to remove that as well. After that, the categorical variables `Survived`, `Sex`, and `Embarked` are transformed as factors:

```
tbl_cleaned <- tbl %>%
  select( -Cabin, -Name, -Ticket ) %>%
  mutate(
    Survived = as.factor(Survived),
    Sex = as.factor(Sex),
    Embarked = as.factor(Embarked)
  )
```

At this point, it is possible to calculate the Pearson correlation for each pair of numerical variables by applying the pooling technique provided by Rubin in multiple imputations. The `miceadds` package exposes wrapper functions that simplify this operation for the most common statistical analysis given the result of the `mice()` function as a parameter. In our case, the function of interest is `micombine.cor()`, and we use it in our `corr_impute_missing_values()` function:

```
corr_impute_missing_values <- function(df, m = 5, variables, method = c('pearson', 'spearman')) {
  method <- method[1]
  df_imp_1st <- mice(df, m = m, printFlag = FALSE)
  corr_tbl <- miceadds::micombine.cor(df_imp_1st, variables = variables, method = method) %>%
    as_tibble() %>%
    arrange(variable1, variable2)
  return(corr_tbl)
}
```

It is easy therefore to obtain the aforementioned correlations:

```
# Get the indexes of numeric columns
numeric_col_idxs <- which(sapply(tbl_cleaned, is.numeric))
corr_tbl <- corr_impute_missing_values(tbl_cleaned, variables = numeric_col_idxs, method = 'pearson')
corr_tbl
```

Here's the result:

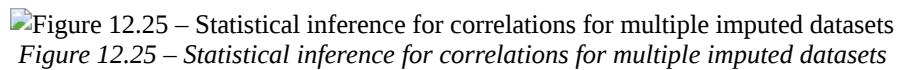


Figure 12.25 – Statistical inference for correlations for multiple imputed datasets

Without going into too much detail about the other fields, the correlation coefficient between the variables is given by the `r` column. Since the `r` coefficient is the result of a process of inference, the `lower95` and `upper95` columns define the upper and lower 95% confidence interval bounds.

Important Note

If you get an error such as **Error in matchindex(yhatobs, yhatmis, donors) : function 'Rcpp_precious_remove' not provided by package 'Rcpp'**, it is likely that you are running a recent version of a package compiled with an earlier version of `Rcpp`. Updating `Rcpp` with the `install.packages('Rcpp')` command should fix it.

Sometimes, the goal of your analysis is not to get results from statistical functions, but to simply fill in the holes left by missing values because the dataset in question must then be used to train a machine learning algorithm that does not admit null values. The latest versions of scikit-learn (currently in the experimental phase) expose the `impute` module with its `SimpleImputer`, `KNNImputer`, and `IterativeImputer` methods. In this way, it is possible to impute the missing values of a dataset through machine learning algorithms (k-nearest neighbors; linear regression) among other more naïve methods (substitutions with fixed values, mean, median, or mode), and also to have an average score of how the algorithm performs in general (cross-validated mean squared error). You'll see an example of one of these methods in *Chapter 13, Using Machine Learning without Premium or Embedded Capacity*.

If, on the other hand, you need to impute missing values from a univariate time-series, how would you proceed? Let's see it.

Imputing time-series data

You can find the code used in this section in the `Chapter12\R\05-handle-time-series-missing-values-in-r.R` file.

We consider a time-series of the average number of aircraft passengers per month. Let's duplicate it and eliminate from it 10% of the values randomly and, in addition, let's eliminate a couple of duplicated values by hand. Then, we'll merge the two time-series into a single tibble:

```

air_df <- read.csv('https://bit.ly/airpassengers')
# Create 10% of missing values in the vector
set.seed(57934)
value_missing <- missForest::prodNA(air_df['value'], noNA = 0.1)
# Force a larger gap in the vector
value_missing[67:68, ] <- NA
# Add the vector with missing values to the dataframe
air_missing_df <- air_df %>%
  mutate( date = ymd(date) ) %>%
  rename( complete = value ) %>%
  bind_cols( value = value_missing )

```

The result can be seen in *Figure 12.22*. The `imputeTS` package exposes convenient functions that implement the missing value imputations already described in the *Univariate time-series imputation* section. Once the values are imputed using the different algorithms and parameters, it is possible to calculate the accuracy because you also know the complete time-series. We use the `accuracy()` function exposed by the `forecast` package to calculate the final accuracy using various metrics, such as *mean absolute error* and *root mean squared error*:

 *Figure 12.24 – Error metrics for imputed values in a time-series*
Figure 12.24 – Error metrics for imputed values in a time-series

The **Seasonally Decomposed Imputation (seadec)** strategy seems to be the best one. Here's the plot of missing values according to this strategy:

 *Figure 12.25 – Representation of imputed values in the time-series*
Figure 12.25 – Representation of imputed values in the time-series

Let's now look at how to use what we've learned so far about missing values in Power BI.

Imputing missing values in Power BI

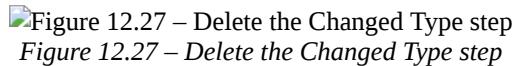
We have delved into the theory and techniques of imputing missing values, whether you are dealing with a tabular dataset or a time-series, precisely because in Power BI there is no way to adopt them through native tools, except for the naïve solution of replacing them with a default value (such as fixed value, mean, or median). In fact, when the business analyst finds themselves needing to fill in the gaps in the data, they often ask for the help of a data scientist or someone with the statistical knowledge to tackle the problem. Now that you've studied this chapter, you're able to tackle it on your own!

Let's apply what we did in the previous sections for tabular and time-series data in Power BI:

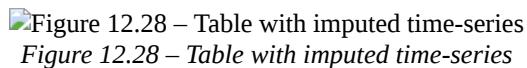
1. Open Power BI Desktop and make sure it is referencing the latest engine.
2. Click **Get data**, search for **web**, and double-click on the **Web** connector.
3. Enter the following URL and click **OK**: <http://bit.ly/titanic-dataset-csv>.
4. On the next import screen, click **Transform Data**.
5. Go to the **Transform** tab, click on **Run R script**, copy the script in the `06-impute-tabular-missing-values-in-power-bi-with-r.R` file in the `Chapter12\R` folder, paste it into the editor, and then click **OK**.
6. You may be asked to configure the privacy levels of the R script and the CSV file. In this case, select both the **Organizational** and **Public** level.
7. We are only interested in the data in `corr_tbl`. So, click on its **Table** value.
8. As a result, you'll see the table containing the correlation coefficients calculated using MICE and the pooling method provided by the multivariate imputation technique:

 *Figure 12.26 – Correlation table calculated with the multivariate imputation technique*
Figure 12.26 – Correlation table calculated with the multivariate imputation technique

9. Go to the **Home** tab and click **Close & Apply**.
10. Click **Get data** and double-click on the **Text/CSV** connector.
11. Select the `air.csv` file you can find in the `Chapter12` folder and click **Open**.
12. On the next import screen, click **Transform Data**.
13. Power BI automatically interprets the `date` text field as a date field, and therefore applies a **Changed Type** operation from Text to Date. In order to correctly process dates within an R script with the `lubridate` package, you must delete the **Changed Type** step by clicking on the red cross before inserting the R script:


Figure 12.27 – Delete the Changed Type step

14. Go to the **Transform** tab, click on **Run R script**, copy the script in the `07-impute-time-series-missing-values-in-power-bi-with-r.R` file in the `chapter12\R` folder, paste it into the editor, and then click **OK**.
15. You may be asked to configure the privacy levels of the R script and the CSV file. In this case, select both the **Organizational** and **Public** level.
16. As a result, you'll see the table containing the original time-series (the `value` column) and other time-series obtained through different imputation algorithms (each in a different column):


Figure 12.28 – Table with imputed time-series

17. Go to the **Home** tab and click **Close & Apply**.

Impressive! You managed to apply the most complex missing value imputation algorithms to a tabular dataset and a time-series in Power BI with minimal effort. Congratulations!

Summary

In this chapter, you learned what outliers are, what causes them generally, and how they are treated. You also learned how to identify them based on the number of variables involved and their given type, both in Python and in R.

Another important topic you covered was how to impute missing values in tabular and time-series datasets. You learned how to diagnose them and impute them with R.

After that, you implemented the value imputation algorithms in Power BI.

In the next chapter, you will see how to use machine learning algorithms in Power BI without the need for Premium or Embedded capabilities.

References

For additional reading, check out the following books and articles:

1. Add Marginal Plot to ggplot2 Scatterplot Using ggExtra Package in R (<https://statisticsglobe.com/ggplot2-graphic-with-marginal-plot-in-r>)
2. 5 Things You Should Know About Covariance (<https://towardsdatascience.com/5-things-you-should-know-about-covariance-26b12a0516f1>)
3. Mahalanobis Distance and its Limitations (<https://rpubs.com/jjsuarezstra99/mahalanobis>)
4. Box-Cox Transformation Explained (<https://towardsdatascience.com/box-cox-transformation-explained-51d745e34203>)

5. How to Use Power Transforms for Machine Learning (<https://machinelearningmastery.com/power-transforms-with-scikit-learn/>)
6. The Relationship between the Mahalanobis Distance and the Chi-Squared Distribution (<https://markusthill.github.io/mahalanobis-chi-squared/>)
7. Using the recipes package for easy pre-processing (http://www.rebeccabarter.com/blog/2019-06-06_pre_processing/)
8. Anomaly detection (<https://docs.microsoft.com/en-us/power-bi/visuals/power-bi-visualization-anomaly-detection>)
9. Missing data: mechanisms, methods, and messages (http://www.i-deel.org/uploads/5/2/4/1/52416001/chapter_4.pdf)
10. Multiple imputation by chained equations: what is it and how does it work? (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3074241/>)
11. Amelia II: A Program for Missing Data (<https://www.jstatsoft.org/article/view/v045i07>)
12. All about Categorical Variable Encoding (<https://towardsdatascience.com/all-about-categorical-variable-encoding-305f3361fd02>)

13 Using Machine Learning without Premium or Embedded Capacity

Thanks to the computing power now available via powerful laptops or through the cloud, you can enrich your analysis with insights from machine learning models easily and instantly. **Power BI** provides integrated tools (closely related to data flows) that allow you to use machine learning models developed by data scientists on Azure Machine Learning, models trained and deployed through **Azure AutoML**, or services exposed by cognitive services directly through a convenient graphical interface. The only drawback is that these tools (known as **Advanced AI**) are only enabled if you use an **Embedded** capacity, Premium capacity, or **Premium Per User (PPU)** license. Does this mean that a user using Power BI Desktop or simply the Power BI service with a Pro license cannot benefit from machine learning? Absolutely not, and we'll show you how to do it thanks to **Python** and **R**.

In this chapter, you will cover the following topics:

- Interacting with ML in Power BI with data flows
- Using AutoML solutions
- Embedding training code in Power Query
- Using trained models in Power Query
- Using trained models in Script Visuals
- Calling web services in Power Query

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2*,

Configuring R with Power BI, and Chapter 3, Configuring Python with Power BI.

Interacting with ML in Power BI with data flows

You can access **Advanced AI features** directly through Power BI Desktop or you can access **Advanced AI features for dataflow** through data flows, which are easy-to-use tools for transforming big data into insights to be shown in dashboards. But, as you can imagine, both modes require the aforementioned licenses in the introduction.

These features are accessible from Power BI Desktop, in the **Power Query Home** ribbon:

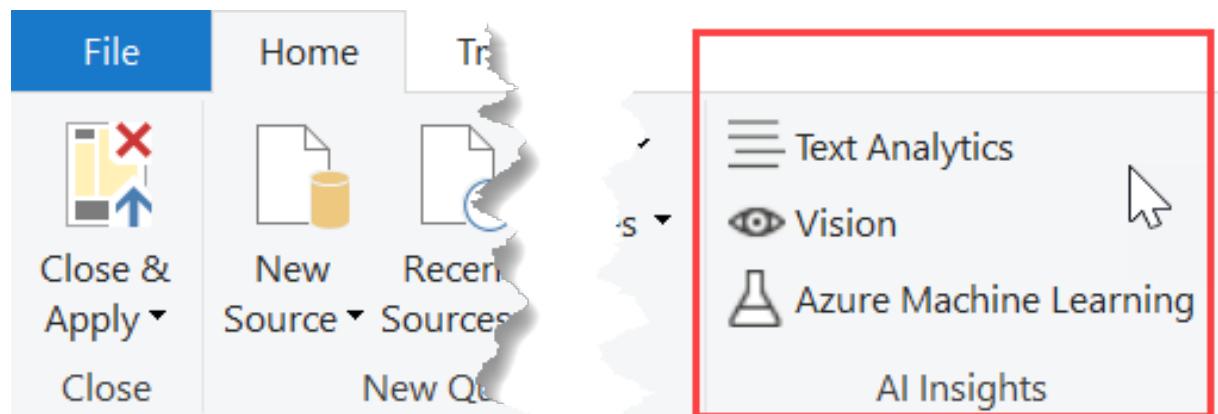


Figure 13.1 – AI insights in Power BI Desktop

The first two options (**Text Analytics** and **Vision**) you can see in *Figure 13.1* use **Azure Cognitive Services** behind the scenes, specifically Text Analytics services and Computer Vision services. Basically, thanks to these features in Power BI, you can now use *four functions* to enrich your data through the power of machine learning.

AI insights

The screenshot shows the Power BI AI Insights interface. At the top is a search bar with a magnifying glass icon and the placeholder text "Search". Below the search bar is a yellow header bar with a folder icon and the text "Cognitive Services [4]". Underneath the header, there are four function entries, each with an "fx" icon: "CognitiveServices.TagImages", "CognitiveServices.ExtractKeyPhrases", "CognitiveServices.DetectLanguage", and "CognitiveServices.ScoreSentiment".

- fx CognitiveServices.TagImages
- fx CognitiveServices.ExtractKeyPhrases
- fx CognitiveServices.DetectLanguage
- fx CognitiveServices.ScoreSentiment

Figure 13.2 – Cognitive Services functions in Power BI

These are as follows:

- **TagImages.** Analyzes images to generate tags based on what they contain
- **ExtractKeyPhrases.** Evaluates unstructured text, and for each text column, returns a list of key phrases
- **DetectLanguage.** Evaluates text input, and for each column, returns the language name and ISO identifier
- **ScoreSentiment.** Evaluates text input and returns a sentiment score for each document, ranging from 0 (negative) to 1 (positive)

The other option of AI Insights is to be able to use models hosted in **Azure Machine Learning** as scoring functions in Power Query.

Azure Machine Learning Models

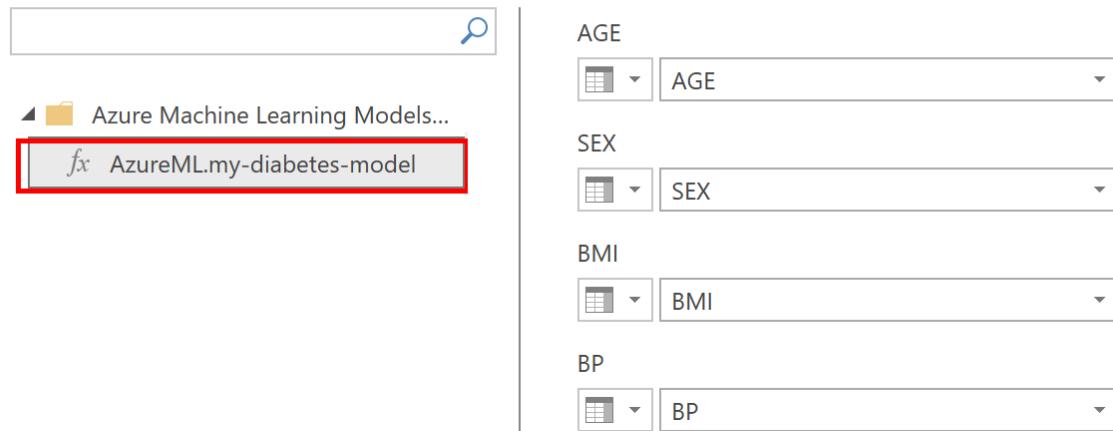


Figure 13.3 – Azure Machine Learning functions in Power BI

To top it off, the Advanced AI features also include the ability to create machine learning models on the fly via a GUI thanks to **AutoML for dataflows**.

AutoML solutions are very convenient, especially for the analyst who doesn't have much experience with machine learning. You will see this in detail in the next section. Now you just need to know that in Power BI, you can generate three types of models: **classifications** (binary or multi-label), **regressions**, and **time series forecasting** (will be available soon).

Choose a model type

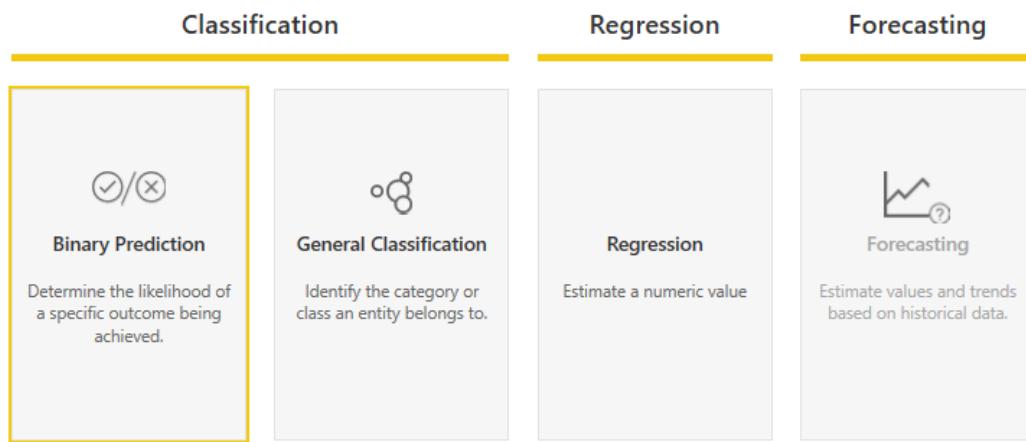


Figure 13.4 – AutoML for data flows in Power BI

Behind the scenes, there is the Azure AutoML service that allows you to do model training, but by leveraging data flows, you don't need to instantiate a machine learning workspace to run AutoML experiments.

A user who only has a Power BI Pro license cannot access these fantastic features directly from the Power BI GUI. However, thanks to the introduction of Python and R in Power BI, it is possible to use machine learning algorithms or external services that facilitate their implementation with just a few lines of code.

Is it really possible that just a few lines of code are enough to train a machine learning model? Where is the trick!? Let's explain the mystery.

Using AutoML solutions

Writing code from scratch to do machine learning requires specific knowledge that a generic analyst using Power BI often doesn't know. Therefore, we recommend the use of **Automated Machine Learning (AutoML)** processes from here on out for analysts who do not have a data science background. Does this mean that anyone can create an accurate machine learning model without knowing the theory behind this science

simply by using AutoML algorithms? Absolutely not! The following applies:

Important Note

An AutoML tool relieves the analyst of all those repetitive tasks typical of a machine learning process (hyperparameter tuning, model selection, and so on). Often, those that require specific theoretical knowledge on the part of the analyst (for example, missing value imputation, dataset balancing strategies, feature selection, and feature engineering) are left out of the automated steps. Therefore, not applying the appropriate transformations that only an expert knows to the dataset before starting an AutoML process leads to the generation of a baseline model that might be sufficiently accurate, but could not ensure product performance.

You might think that AutoML tools are hated by data scientists. This is also a myth. Many of them use it as a quick and dirty prototyping tool and as an executor of repetitive steps while they focus on more critical tasks.

In this chapter, we will be satisfied with obtaining discrete performance models (sometimes very good if we are lucky enough to have a properly transformed training dataset), and so the outputs provided by AutoML solutions are more than fine.

Moreover, we will exclusively use AutoML solutions in Python as it is the most widely used language in most third-party machine learning platforms. The R language is a little less widely used than Python, but that doesn't mean the results you get in R are any less valid. On the contrary, as you may have noticed in previous chapters, some specialized packages of statistical functions that allow high flexibility of data manipulation exist only for R and not for Python.

Simply put, working with machine learning in Python to date allows models to be easily shared between popular platforms. Therefore, we suggest it for Power BI analysts, who perhaps prefer to delegate model creation to more specialized platforms and then import them in Power BI.

Let's see what AutoML tools we'll be using in the code for this chapter.

PyCaret

PyCaret (<https://pycaret.org/>) is an open source, low-code machine learning library in Python that automates the cycle of machine learning experiments, democratizing access to these advanced techniques to business analysts and domain experts, and also helping data scientists to become more efficient and productive.

The types of problems that PyCaret can solve are as follows:

- **Classification** (predicting a categorical target variable)
- **Regression** (predicting a numeric target variable)
- **Clustering** (grouping of observations into specific sets, each with its own properties)
- **Anomaly Detection** (the process of finding outliers in a dataset, and which are far fewer in number than with usual observations)
- **Natural Language Processing** (text transformations in useful features for classifications and regressions)
- **Association Rules** (a rule-based technique that finds important relationships between features according to probability theory)

For more experienced users, PyCaret also provides convenient functions for model ensembling and model explanation.

Azure AutoML

Azure AutoML is a cloud-based service that can be used to automate the construction of machine learning pipelines for classification, regression, and forecasting tasks. Such pipelines involve a pre-processing phase of the dataset to better fit the training algorithms used in the next phase. After tuning and training multiple models, Azure AutoML selects the most accurate model among them, while also considering two other models resulting from the ensembling of the previously trained models.

The available tasks are as follows:

- Classification
- Regression
- Time series forecasting

For more detailed coverage of this platform, please take a look at the references.

RemixAutoML for R

For the sake of completeness, we also suggest one of AutoML's solutions for R, which is **RemixAutoML**

(<https://github.com/AdrianAntico/RemixAutoML>). It is a set of functions that facilitate the use of many AutoML packages available for R (CatBoost, LightGBM, XGBoost, and H2O). In addition to giving the inexperienced analyst the ability to create machine learning models with a few lines of code thanks to AutoML, this library also contains very advanced features (for example, features for feature engineering and time series forecasting), often used by more experienced analysts.

Let's now look at the various ways to use machine learning models in Power BI.

Embedding training code in Power Query

One of the easiest solutions to train a machine learning model is to write the code needed to do so directly in Power Query, right after importing a dataset on which you will build the model.

Training a model on a fairly large dataset typically takes quite a bit of time to complete. As you embed the code in Power Query, it will run every time the data is refreshed, and this may result in a non-negligible delay in getting the data online. Hence, the following applies:

Important Note

This solution is recommended when you are certain that the time required to complete the model training is acceptable.

Let's now look at an example of how to write some training code using PyCaret.

Training and using ML models with PyCaret

Let's take the Titanic disaster dataset to train a machine learning model. Specifically, we want to create a model that predicts whether a passenger survives (the `Survived` column) based on their attributes described by the other features in the dataset. Evidently, this is a *binary classification* (does it survive? yes or no) that we can easily implement with PyCaret.

As PyCaret is constantly evolving and so are all the other dependent libraries, you need to also install the **Visual C++ Build tools** to build the necessary wheels and avoid errors such as *Failed building wheel for <package>*. Here are all the steps needed to install PyCaret correctly for Windows:

1. Download the installer from <https://visualstudio.microsoft.com/visual-cpp-build-tools/> and run it.
2. In the next window, check just the **Desktop development with C++** option.
3. You will be prompted to restart the machine. Do so.

4. Once your laptop has restarted, run your Anaconda Prompt and enter the following command to create the new `pycaret_env` environment:

```
conda create --name pycaret_env python=3.7
```

5. Enter the following command to switch to the new environment:

```
conda activate pycaret_env
```

6. Enter the following command to install the full version of PyCaret:

```
pip install pycaret
```

Having done that, you can move on to see the training code of the model. The only little hiccup is the handling of missing values in the dataset (you already had a chance to analyze them in *Chapter 12, Adding Statistics*

Insights, Outliers, and Missing Values). Unfortunately, PyCaret currently only supports handling missing values using the simplest methods, namely, imputation using the mean or median for numeric values, and imputation using the mode or a fixed string for categorical values. Since we want to show you how to impute missing values using the *K-Nearest Neighbors (KNN)* algorithm as anticipated in *Chapter 12, Adding Statistics Insights, Outliers, and Missing Values*, you will write a few more lines of code than usual.

The code used to impute the missing values via the KNN algorithm will be used in the first transformation step in Power BI, after importing the data from the Titanic dataset. You can find the code in the

`01-impute-dataset-with-knn.py` file in the `Chapter13/Python` folder. It will take care first to operate a simple feature selection, eliminating those fields that could cause noise during the training of the model. After that, since the above imputation algorithm exposed by scikit-learn via the **KNNImputer** module does not handle categorical variables in the dataset, the code also takes care of doing the encoding of the categorical variables using the **ordinal encoding** technique (using a mapping of categories to integers) thanks to the **OrdinalEncoder** module of scikit-learn. At this point, the code imputes the missing values using the default distance measure, that is, a Euclidean distance measure that will not include NaN values when calculating the distance between members of the training dataset.

Once the imputed dataset is available, you can train the model with which you will then score a test dataset. You can find the code in the

`02-train-model-with-pycaret.py` file in the `Chapter13/Python` folder. For convenience, you will use 95% of the imputed dataset to train the model, while the remaining 5% will be used as a test dataset. All this will go in a transformation step following the previous one used for the imputation of missing values in Power BI.

You'll split the dataset into training and test sets thanks to scikit-learn's `train_test_split()` method. After that, the model training is done very simply by calling PyCaret's `setup()` and `compare_models()` functions. In the `setup()` function, you will define which dataframe to use for training,

the target variable (`Survived`), and which are the categorical and which are the ordinal variables. Moreover, it is necessary to use silent mode, otherwise user intervention would be required to validate the inferred types of the other variables. The `compare_models()` function trains and evaluates the performance of all models provided by PyCaret for classification using cross-validation. In addition to returning the best-performing model, this function also returns the performance values of each model returned by cross-validation.

Model		Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
lightgbm	Light Gradient Boosting Machine	0.8136	0.8519	0.7158	0.8075	0.7543	0.6051	0.6122	0.0080
gbc	Gradient Boosting Classifier	0.8034	0.8454	0.6864	0.8013	0.7340	0.5802	0.5892	0.0150
rf	Random Forest Classifier	0.7881	0.8538	0.7371	0.7399	0.7348	0.5589	0.5628	0.1070
lr	Logistic Regression	0.7814	0.8339	0.6739	0.7575	0.7077	0.5352	0.5422	0.0150
et	Extra Trees Classifier	0.7780	0.8360	0.7114	0.7305	0.7181	0.5353	0.5380	0.1000
ada	Ada Boost Classifier	0.7763	0.8175	0.6859	0.7417	0.7079	0.5277	0.5328	0.0160
lda	Linear Discriminant Analysis	0.7763	0.8329	0.6699	0.7481	0.7023	0.5250	0.5308	0.0030
ridge	Ridge Classifier	0.7746	0.0000	0.6658	0.7471	0.6998	0.5213	0.5269	0.0040
nb	Naive Bayes	0.7576	0.7977	0.6866	0.7050	0.6921	0.4929	0.4964	0.0040
dt	Decision Tree Classifier	0.7424	0.7297	0.6696	0.6889	0.6751	0.4623	0.4661	0.0040
qda	Quadratic Discriminant Analysis	0.7237	0.7409	0.5953	0.7102	0.6067	0.4081	0.4269	0.0040
knn	K Neighbors Classifier	0.6898	0.7214	0.5460	0.6320	0.5848	0.3393	0.3425	0.0260
svm	SVM - Linear Kernel	0.6898	0.0000	0.7136	0.6083	0.6385	0.3757	0.3954	0.0040

Figure 13.5 – Performance of all models

Figure 13.5 shows several typical classification metrics for each model. One of the most widely used is the **Area Under the ROC Curve (AUC or AUC-ROC)** when the dataset is balanced (in other words, when there is a slight disproportion between the number of observations associated with one class of the target variable versus the other class). The following remark applies:

Important Note

The `compare_models()` function doesn't work in Power BI unless you disable the parallelism, passing `n_jobs=1` into the `setup()` function. If you don't assign `1` to `n_job`, by default, PyCaret assigns the value

-1 (maximum parallelism) and behind the scenes, the best model is computed correctly using multiple threads, but Power BI can't trace it back to the main process, so it gets stuck.

With an AUC of about 0.85 (it can vary as the process is stochastic), the **Random Forest Classifier** appears to be the best model obtained by training 95% of the imputed dataset. Then you will use the newly trained model (`best_model`) to obtain predictions of the remaining 5% of the dataset via PyCaret's `predict_model()` function. You will get a result similar to this one:

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	Survived	Label	Score
0	2.0	1.0	32.4	0.0	0.0	13.0000	2.0	1.0	1.0	0.7764
1	3.0	1.0	28.0	0.0	0.0	9.5000	2.0	0.0	0.0	0.9100
2	3.0	0.0	27.0	0.0	0.0	7.9250	2.0	1.0	1.0	0.8400
3	1.0	0.0	58.0	0.0	1.0	153.4625	2.0	1.0	1.0	0.9900
4	3.0	1.0	28.0	1.0	0.0	15.8500	2.0	0.0	0.0	0.9900
...
839	3.0	0.0	29.0	1.0	1.0	10.4625	2.0	0.0	0.0	0.9300
840	3.0	0.0	17.2	0.0	0.0	7.2292	0.0	1.0	1.0	0.9700
841	1.0	1.0	34.0	0.0	0.0	26.5500	2.0	1.0	1.0	0.9600
842	2.0	1.0	19.0	0.0	0.0	10.5000	2.0	1.0	0.0	0.5717
843	3.0	0.0	21.0	0.0	0.0	7.7500	1.0	0.0	0.0	0.7700

Figure 13.6 – Predictions of the test dataset

As you can see, the result generated by scoring the dataset consists of two new columns for classification: the `Score` column represents an estimate of a measure, such as the probability with which the predicted classes are those reported in the `Label` column. If you are interested in having a true probability estimate, you have to **calibrate** the model (have a look at the references for more details). The trained model will also be saved as a `PKL` file for future reuse.

Let's look at how to implement what's explained up here in Power BI.

Using PyCaret in Power BI

First, make sure that Power BI Desktop references the new `pycaret_env` Python environment in **Options**. Then, follow these steps:

1. Click on **Get Data**, search for `web`, select **Web**, and then click on **Connect**.
2. Enter the `http://bit.ly/titanic-dataset-csv` URL into the URL textbox and click **OK**.
3. You'll see a preview of the data. Then, click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.
5. Enter the script you can find in the `01-impute-dataset-with-knn.py` file in the `Chapter13\Python` folder.
6. We are only interested in the data in the `df_imputed` dataframe. So, click on its **Table** value.
7. You'll see a preview of the dataset with all the missing values imputed.
8. Click **Transform** on the ribbon and then **Run Python script**.
9. Enter the script you can find in the `02-train-model-with-pycaret.py` file in the `Chapter13\Python` folder.
10. We are only interested in the data in the `predictions` dataframe. So, click on its **Table** value.
11. You'll see a preview of the dataset with the predictions generated by the model and the input dataset.
12. Click **Home** on the ribbon and then click **Close & Apply**.

Amazing! You have just trained a machine learning model and then scored a test dataset using a few lines of Python code thanks to PyCaret!

Let's now see how to proceed when the model is trained outside of Power BI.

Using trained models in Power Query

As you already saw in *Chapter 4, Importing Unhandled Data Objects*, you used to share objects that were the result of complex, time-consuming processing (thus also a machine learning model) in a serialized format specific to the language you were using. At that point, it was very simple to deserialize the file and get the model ready to be used in Power Query to predict the target variable of new observations. However, it is important to know the dependencies needed by the scoring function (which gets the new observations as input and returns the predictions), since they are closely related to how the training of the model took place. For this reason, we recommend the following:

Important Note

When you need to use a serialized machine learning model provided by a third party, make sure that whoever developed it also provides you with a working scoring function in order to avoid unnecessary headaches when predicting target values for unknown observations.

If you think about it, the ability to serialize and deserialize a machine learning model could somehow solve the delay problem raised in the case of training the model directly in Power Query in the previous section. Suppose you run the embedded training code for the first time. Immediately afterward, you serialize the model and save it to disk. On the next refresh, instead of running the training code again, you can check whether the serialized model file exists in the expected path. If yes, you load the file, deserialize it, and use that model for the next steps; otherwise, you run the training code again.

Evidently, the aforementioned process involves the intervention of an expert who decides to eliminate the serialized file when the model does not perform very well because perhaps the business data has, in the meantime, changed substantially such that the previous model is not accurate any longer, like it was after the training undertaken with the past data (a process known as **model drift**; take a look at the references for more details).

We will not go into the implementation details of this solution, but we wanted to provide just a tip for a possible solution to the problem raised in the previous section.

Let's now implement the scoring of a dataset of unseen observations in Power BI using an already trained PyCaret model.

Scoring observations in Power Query using a trained PyCaret model

If you remember correctly, in the previous section, you saved the model trained in Power BI to a PKL file on disk. You also exported the test dataset calculated in the same code to CSV. In this session, you will directly use the serialized model, loading it with the `load_model()` function, and the test CSV dataset to be scored in Power BI. Since the model was trained using PyCaret, the scoring function to use is simply given by the `predict_model()` function. Keep in mind that the scoring function may be more complex when not using a framework such as PyCaret that simplifies things.

These are the steps to follow in Power BI:

1. Click on **Get Data**, select **Text/CSV**, and then click on **Connect**:
2. Select the `titanic-test.csv` file in the `Chapter13` folder and click **Open**.
3. You'll see a preview of the test data. Then, click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.
5. Enter the script you can find in the `03-score-dataset-using-pycaret-model.py` file in the `Chapter13\Python` folder.
6. We are only interested in the `predictions` dataframe. So, click on its **Table** value.
7. You'll see a preview of the test dataset with two additional columns – `Label` and `Score`.
8. Click **Home** on the ribbon and then click **Close & Apply**.

As you can see, this is the most immediate and common way to use a custom machine learning model for scoring in Power BI. In fact, we recommend the following:

Important Note

It is convenient to do the training and, in general, to manage a machine learning model in platforms external to Power BI in order to decouple any development/tuning interventions of the model from the rest of the report.

Let's now see instead how to use serialized machine learning models directly in **Script Visuals**.

Using trained models in Script Visuals

As you learned in *Chapter 4, Importing Unhandled Data Objects*, thanks to object serialization and its string representation, you can import any object into a Python or R visual in the form of a dataframe of strings. Once said dataframe is available in the script visual, you can revert it to the original object via inverse deserialization transformations. Since you can do what we described with any object, evidently you can also do it for machine learning models already trained outside of Power BI.

When the appropriately deserialized model is available in the *Script Visual* session, new observations can be predicted immediately using the scoring function described in the previous section.

The first thing you might ask yourself is what's the point of being able to score a dataset inside a script visual when the data must always be available first in the Power BI data model in order to use it in the visual. In fact, if the data of the observations to use as input to the model is already found in the data model of Power BI, it could be better to apply batch scoring directly in Power Query and so use the predictions as a new column of the dataset. All of this is absolutely true. However, there are some cases in which it is convenient to use a script visual:

Important Note

It is convenient to use a machine learning model in a script visual when you need to realize some simulation reports that allow you to explore the outputs of the model and vary the variables in play dynamically without having to refresh the entire report.

In this case, we suggest using **What-If parameters** (<https://bit.ly/power-bi-what-if>) in Power BI for numeric features, which are dynamic and give the user a very usable report. For categorical variables, you can enter their content manually in Power BI using the **Enter Data** feature, which creates a disconnected table. What-If parameters create disconnected tables by default in the data model.

To properly understand this paragraph, make sure you understand the content of *Chapter 4, Importing Unhandled Data Objects*. Suppose you have to provide observations to a machine learning model that expects two variables as input – a numeric one and a categorical one. When passing the information to the script visual's dataframe, in addition to the fields of the serialized model's dataframe (`model_id`, `chunk_id`, and `model_str`) coming from Power Query, you will also have to assign the associated values to both the parameter slicers related to the two input variables. Since only one value is selected at a time for each parameter when slicing, the set of all the parameters form a tuple, which in our case is (`numeric_value`, `category_id`). This tuple will be replicated as many times as the rows of the string chunk dataframe (consisting of the columns `model_id`, `chunk_id`, and `model_str`), and concatenated to it in order to provide the final dataframe that will be available in the variable named `dataset` in the Script Visual session.

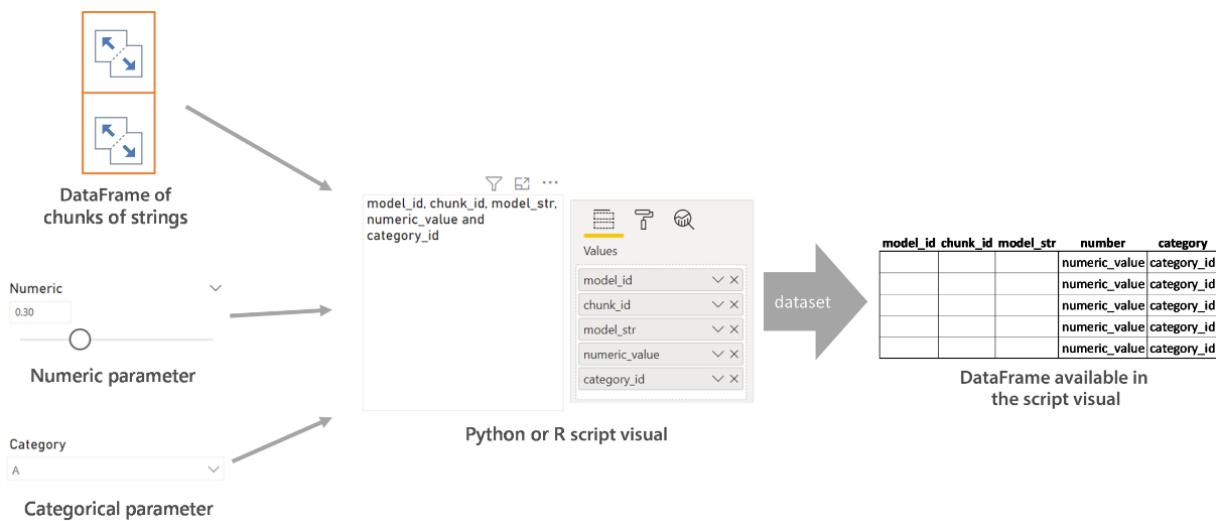


Figure 13.7 – Deserializing the PKL file content into a Python visual

Once you have the dataset dataframe available in the script visual, you can apply the deserialization transformations just to the columns (`model_id`, `chunk_id`, and `model_str`), thereby obtaining the machine learning model ready to be used. Selecting instead just the columns (`number`, `category`) and applying the distinct function to all the rows of the resulting dataframe, you obtain back the tuple of parameters to provide by way of input to the deserialized model. You can therefore calculate the predicted value from the model, providing to it the tuple of parameters as input. You can then use the prediction in the graph to be shown in the script visual.

Let's see in practice how to dynamically predict values from a machine learning model in a Python Script Visual.

Scoring observations in a script visual using a trained PyCaret model

The first thing you will do is to serialize properly the machine learning models (in our case, only one) contained in a dictionary in Power Query. In this way, a dataframe containing the representation in strings of every serialized model of the aforementioned dictionary is obtained. So, it is possible to select the model of interest through a slicer in the report and to therefore use the respective portion of a dataframe in a Python script visual, inside which it will be possible to deserialize the content of the dataframe and to thereby obtain the model to use for scoring.

So, let's proceed to develop our report in Power BI. Make sure that Power BI correctly references the `pycaret_env` environment in **Options**. Here are the steps to follow:

1. Click on **Get data** and then **More....** Start typing `script` into the search textbox and double-click on **Python script**. The Python Script editor will pop up.
2. Copy the content of the `04-serialize-ml-models-in-power-query.py` file into the `Chapter13\Python` folder. Then, paste it into the Python Script editor, changing the absolute path to the PKL file accordingly, and then click **OK**.

3. The navigator window will open, giving you the option to select which dataframe to import. Select both the `model_ids_df` dataframe (containing the model IDs) and the `models_df` one (containing the string representation of serialized models) and then click **Load**. Behind the scenes, a `1:*` relationship is automatically created between the model IDs and the serialized model dataframe via the `model_id` fields.

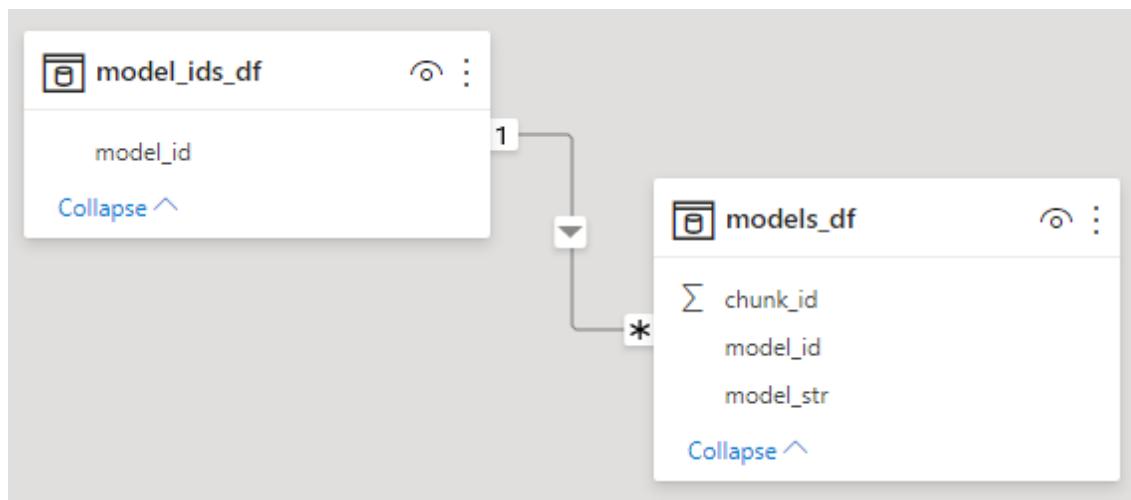


Figure 13.8 – Relationship automatically created between model tables

This relationship allows you to filter the set of rows in the `models_df` table to be used in the Python visual, corresponding to the ID of the model you select via the slicer you'll create in the next step.

4. Click on the slicer visual icon.

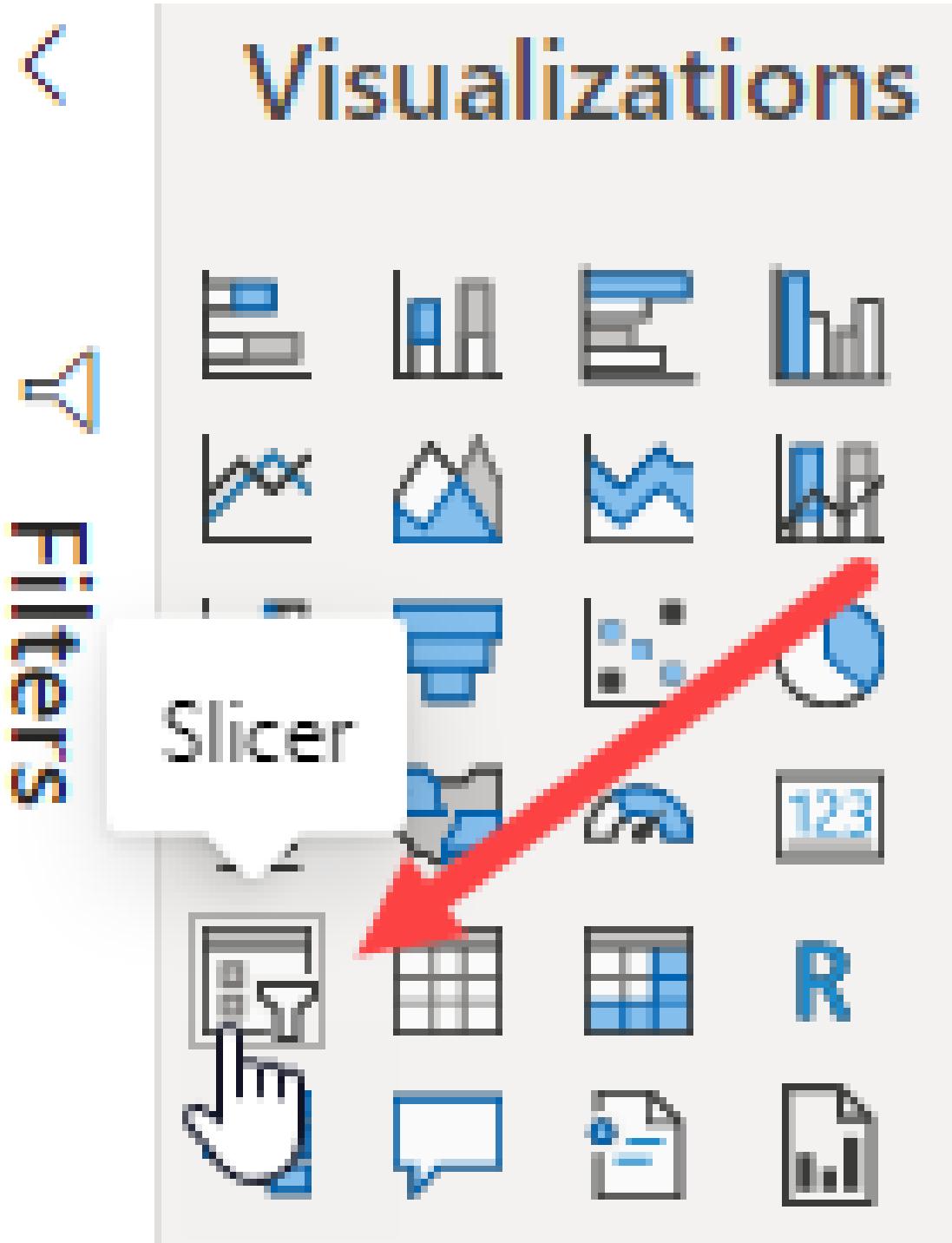


Figure 13.9 – Selecting the slicer visual

Then, click on the **model_id** measure of the **model_ids_df** table.

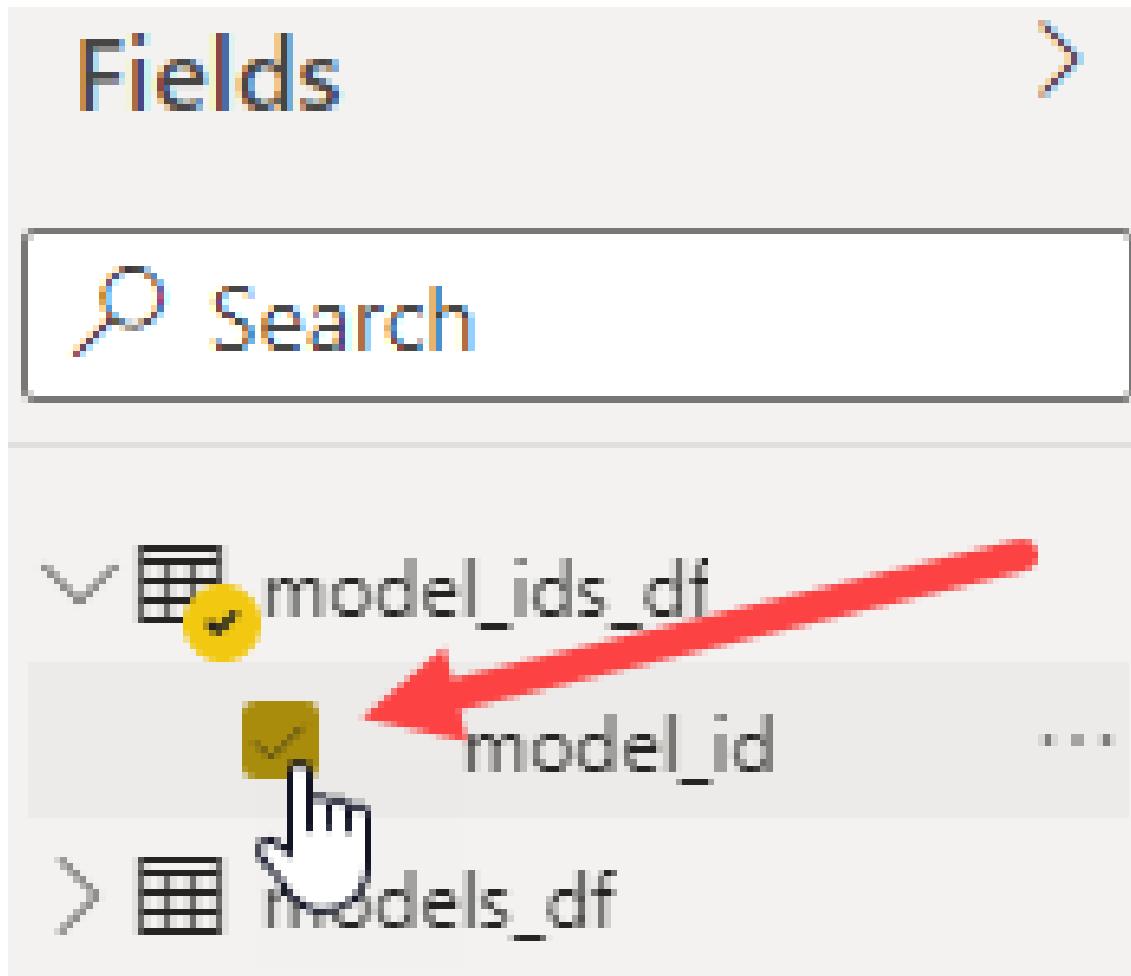


Figure 13.10 – Click on the model_id measure to show it in the slicer

5. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.



Figure 13.11 – Selecting the Dropdown slicer type

6. Resize the bottom edge of the slicer, click on its format options, switch on the **Single select** one, switch off **Slicer header**, and then add the title **Model IDs**.



Figure 13.12 – Setting the slicer options

Figure 13.12 – Setting the slicer options

Then, move it to the top center of the report.

7. You will now add a set of What-If parameters with their slicers associated with each variable to be passed as input to the model. Click on the **Modeling** tab on the ribbon and then on **New parameter**.
8. On the next dialog, enter **Pclass param** in the **Name** field, leave the data type as **Whole number**, enter **1** in the **Minimum** field, **3** in the **Maximum** field, leave **Increment** as **1**, and enter **2** in the **Default** field.

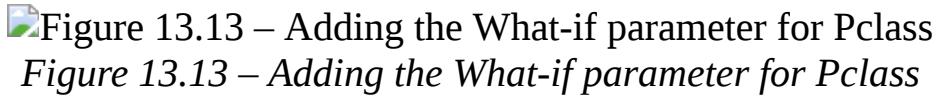


Figure 13.13 – Adding the What-if parameter for Pclass

Figure 13.13 – Adding the What-if parameter for Pclass

Keep **Add slicer to this page** selected and then click **OK**.

9. Resize the bottom edge of the Pclass slicer. Then, click on its format options, switch off **Slicer header**, switch on **Title**, and enter **Passenger class** as the text. Then move it to the top left of your report.
10. Be sure to rename the **Pclass** value of **Pclass param** to the same name as the variable that represents it in the model, namely, **Pclass**.

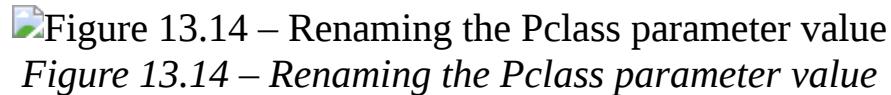


Figure 13.14 – Renaming the Pclass parameter value

Figure 13.14 – Renaming the Pclass parameter value

11. As the variable **Sex** is categorical (F or M), you'll create a disconnected table for it manually. So, click on the **Home** tab in the

ribbon and click on **Enter data**.

12. Create the first column, **Sex**, of the table and add the values 0 and 1 to it. Then, create the new column, **SexLabel**, and enter `Female`, where **Sex** is 0, and `Male`, where **Sex** is 1.



Figure 13.15 – Enter data manually for Sex

Figure 13.15 – Enter data manually for Sex

Enter `Sex` as the table name and then click **Load**.

13. Let's add a slicer for the `Sex` variable. Click on an empty spot in the report canvas first. Then, click the slicer visual to add a slicer to the report. Then, click on the **SexLabel** field and then on the **Sex** field (the order is important). Then, click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type. Also, click on its format options, switch on the **Single select** option in the **Selection control** group, switch off the **Slicer header** option, switch on **Title**, and then enter `Sex` as the text. Resize its bottom edge and move it under the **Passenger class** slicer.



Figure 13.16 – A new dropdown slicer for Sex

Figure 13.16 – A new dropdown slicer for Sex

14. Let's create a new **What-if** parameter for the `Age` variable. Click on the **Modeling** tab on the ribbon and then on **New parameter**. On the next dialog, enter `Age param` in the **Name** field, leave the data type as **Whole number**, enter 1 in the **Minimum** field, 80 in the **Maximum** field, leave **Increment** as 1, and enter 30 in the **Default** field. Keep **Add slicer to this page** selected and then click **OK**.
15. Resize the bottom edge of the `Age` slicer. Then, click on its format options, switch off **Slicer header**, switch on **Title**, and enter `Age` as the text. Then, move it to the top left of your report under the `Sex` slicer.

16. Be sure to rename the **Age** value of **Age param** to the same name as the variable that represents it in the model, namely, `Age`.



Figure 13.17 – Renaming the Age parameter value

Figure 13.17 – Renaming the Age parameter value

17. Let's create a new What-if parameter for the `SibSp` variable. Click on the **Modeling** tab on the ribbon and then on **New parameter**. On the next dialog, enter `SibSp param` in the **Name** field, leave the data type as **Whole number**, enter `0` in the **Minimum** field, `8` in the **Maximum** field, leave **Increment** as `1`, and enter `0` in the **Default** field. Keep **Add slicer to this page** selected and then click **OK**.
18. Resize the bottom edge of the `SibSp` slicer. Then, click on its format options, switch off **Slicer header**, switch on **Title**, and enter `Children/spouse aboard` as the text. Then, move it to the top left of your report under the `Age` slicer.
19. Be sure to rename the **SibSp** value of **SibSp param** to the same name as the variable that represents it in the model, namely, `SibSp`.



Figure 13.18 – Renaming the SibSp parameter value

Figure 13.18 – Renaming the SibSp parameter value

20. Let's create a new What-if parameter for the `Parch` variable. Click on the **Modeling** tab on the ribbon and then on **New parameter**. On the next dialog, enter `Parch param` in the **Name** field, and leave **Whole number** as the data type, enter `0` in the **Minimum** field, `6` in the **Maximum** field, leave **Increment** as `1`, and enter `0` in the **Default** field. Keep **Add slicer to this page** selected and then click **OK**.
21. Resize the bottom edge of the `Parch` slicer. Then, click on its format options, switch off **Slicer header**, switch on **Title**, and enter `Parents/children aboard` as the text. Then, move it to the top left of your report under the `SibSp` slicer.
22. Be sure to rename the **Parch** value of **Parch param** to the same name as the variable that represents it in the model, namely `Parch`:



Figure 13.19 – Renaming the Parch parameter value
Figure 13.19 – Renaming the Parch parameter value

23. Let's create a new What-if parameter for the `Fare` variable. Click on the **Modeling** tab on the ribbon and then on **New parameter**. On the next dialog, enter `Fare param` in the **Name** field, select **Decimal number** as the data type, enter `0` in the **Minimum** field, `515` in the **Maximum**, field, enter `1` in the **Increment** field, and then enter `250` in the **Default** field. Keep **Add slicer to this page** selected and then click **OK**.
24. Resize the bottom edge of the `Fare` slicer. Then, click on its format options, switch off **Slicer header**, switch on **Title**, and enter `Fare` as the text. Then, move it to the top left of your report under the `Parch` slicer.
25. Be sure to rename the **Fare** value of **Fare param** to the same name as the variable that represents it in the model, namely, `Fare`.



Figure 13.20 – Renaming the Fare parameter value
Figure 13.20 – Renaming the Fare parameter value

26. Let's add a slicer for the `Embarked` variable. As the `Embarked` variable is categorical (`0`, `1`, or `2`), you'll create a disconnected table for it manually. So, click on the **Home** tab in the ribbon and then click on **Enter Data**.
27. Create the first column, **Embarked** (this name must correspond to that of the model's variable), of the table and then add the values `0`, `1`, and `2` to it. Then, create a new column, **EmbarkedLabel**, and enter `Cherbourg`, `Queenstown`, and `Southampton`, corresponding to `0`, `1`, and `2`, respectively.



Figure 13.21 – Entering data manually for Embarked
Figure 13.21 – Entering data manually for Embarked

Enter PortEmbarkation as the table name and then click **Load**.

28. Let's now add a slicer for the Embarked variable. Click on an empty spot in the report canvas first. Then, click on the slicer visual in order to add a slicer to the report. Click first on the **EmbarkedLabel** field and then on the **Embarked** one (the order is important). Then, click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type. Also, click on its format options. Switch on **Single select** in the **Selection control** group, switch off **Slicer header**, switch on **Title**, and enter Port of embarkation as the text. Resize its bottom edge and move it under the Passenger class slicer.
29. Now, click on an empty spot in the report canvas, then on **Python Visual** in the **Visualizations** field, and enable it when you're prompted to do so. After that, move it to the center of your report.
30. Keeping it selected, click on all the three fields of the `models_df` table (**chunk_id**, **model_id**, and **model_str**).
31. Still keeping the Python visual selected, also click on all measures (the ones with a calculator icon) of all the parameters entered, and on the identifying fields of the categorical variables (the **Embarked** and **Sex** fields). Remember that the names of the measures must necessarily correspond to the names of the variables provided by the model for the report to work. You may need to enable the Python visual again after selecting the measures. You can do this by simply clicking on the yellow button labeled **Select to enable** in the Python visual. At the end of the selection, you should see all the names of the measures plus those of the fields of the `models_df` table inside the Python visual.



Figure 13.22 – Selected measure names visible in the Python visual
Figure 13.22 – Selected measure names visible in the Python visual

32. Now, click on the Python visual's **Format** tab, expand the **Title** area, edit the text with the **Prediction string**, and increase the font size to 28 point.
33. Copy the code of the `05-deserialize-ml-models-in-python-visual.py` file into the `Chapter13\Python` folder and paste it into the Python visual script

editor. Then, click on the **Run script** arrow icon in the top-right corner of the Python script editor. You'll get a prediction (label and score) of whether a person described by the parameters you selected will survive.



Figure 13.23 – Complete prediction simulation report for the Titanic model

Figure 13.23 – Complete prediction simulation report for the Titanic model

Keep in mind that the same report can be made using models trained in R with the same methodology followed here. In fact, for completeness, we added to the repository the `Chapter13\R` folder containing the scripts corresponding to those used in this section, which is useful for obtaining the same results you got here. In these scripts, we trained the model using a predefined algorithm (**Random Forest**) and used the recently introduced **Tidymodels** framework, which makes use of the Tidyverse principles. For further details, refer to the references.

Wow! You've managed to create a dynamic predictive report in Power BI, something few developers can do!

Let's now see how to invoke the AI and machine learning services exposed by Microsoft in Power BI even if you don't have a Premium capability, an Embedded capability, or a PPU license.

Calling web services in Power Query

Another way to interact with machine learning models within Power Query is to invoke web services. As you may already know, a machine learning model can be used to carry out the scoring of many observations in batch mode using a trained model (process described previously). Another option for being able to interact with a machine learning model is to deploy it to a web service so that it can be invoked via REST APIs. You've already learned how to work with external APIs in *Chapter 9, Calling External APIs to Enrich Your Data*. The following applies to external APIs:

Important Note

Remember that you can't consume external services via REST API calls from a Python or R visual because internet access is blocked for security reasons. Therefore, you can only consume these services in Power Query.

As an example, in this section, you'll see how to invoke predictions from a released endpoint via **Azure Machine Learning** and how to use the services exposed by the **Azure Text Analytics** of cognitive services. You could use some M code in Power Query to access these services, although it's not exactly straightforward. Fortunately, SDKs are available, which make it much easier to access the exposed services. These SDKs are developed for Python, so our examples will be exclusively in Python.

Let's first look at how to interact with a model trained using Azure AutoML.

Using Azure AutoML models in Power Query

In this section, you'll first see how to train a machine learning model using the Azure AutoML GUI. After that, you will use the model released on an Azure container instance as a web service in Power BI.

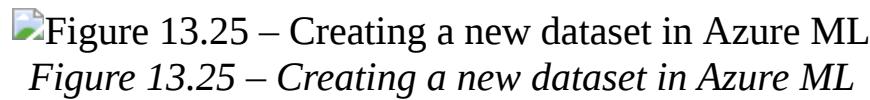
Training a model using the Azure AutoML UI

In order to use Azure AutoML, you must first have access to an Azure subscription (remember you can create a free account as shown at this link: <https://bit.ly/azure-free-account>). After that, you need to create an **Azure Machine Learning Workspace** to train models via the different technologies that Azure provides. You can do this by simply following the steps in the paragraph at this link: <https://bit.ly/create-azureml-workspace>. As soon as the workspace has been allocated, you can log in to **Azure Machine Learning Studio**, an environment in which all the machine learning assets you'll be working with are best organized. Perform the following steps to log in to Azure ML Studio and to start an AutoML experiment:

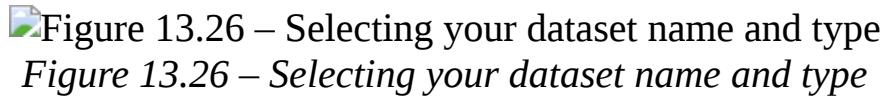
1. Go to <https://ml.azure.com/>.
2. You will be prompted to select an Azure subscription of yours and an Azure ML workspace to work on. Click on **Get started**. You will see something like this:



3. First, you need to import the dataset with which to train the model. You will use the same dataset obtained from the missing value imputation done in the previous sections. Click on **Datasets** in the menu on the left and then on **Create dataset**.



4. You'll be prompted for the dataset name and type. Enter `titanic-imputed` as the name and leave **Tabular** as the type.



Then, click **Next**.

5. You have to upload the CSV file containing the Titanic disaster imputed data. So, click on **Upload**, then on **Upload files**, and finally select the `titanic-imputed.csv` file in the `Chapter13` folder via the **Open file** dialog. The file will be uploaded to the default Azure Blob storage (`workspaceblobstore`) created behind the scenes when instantiating a new Azure ML workspace. Click **Next**.
6. On the next page, you'll get a preview of the dataset you're importing. The engine automatically selects the best import options for you. But if there is something you'd like to change, you can do it on this page. In this case, everything is already OK, so click **Next**.

7. On the following page, you can change the imputed schema of the data you're reading. In this case, leave the inferred type for each field, as the exported CSV file has numeric values with integer and decimal numbers. Then, click **Next**.
8. A recap page will be shown. So, just click **Create** and your dataset will be added to Azure ML.
9. Now you need to create a compute cluster to use for model training. Click on the **Compute** tab on the left menu, then click on **Compute clusters**, and finally click on **New**.

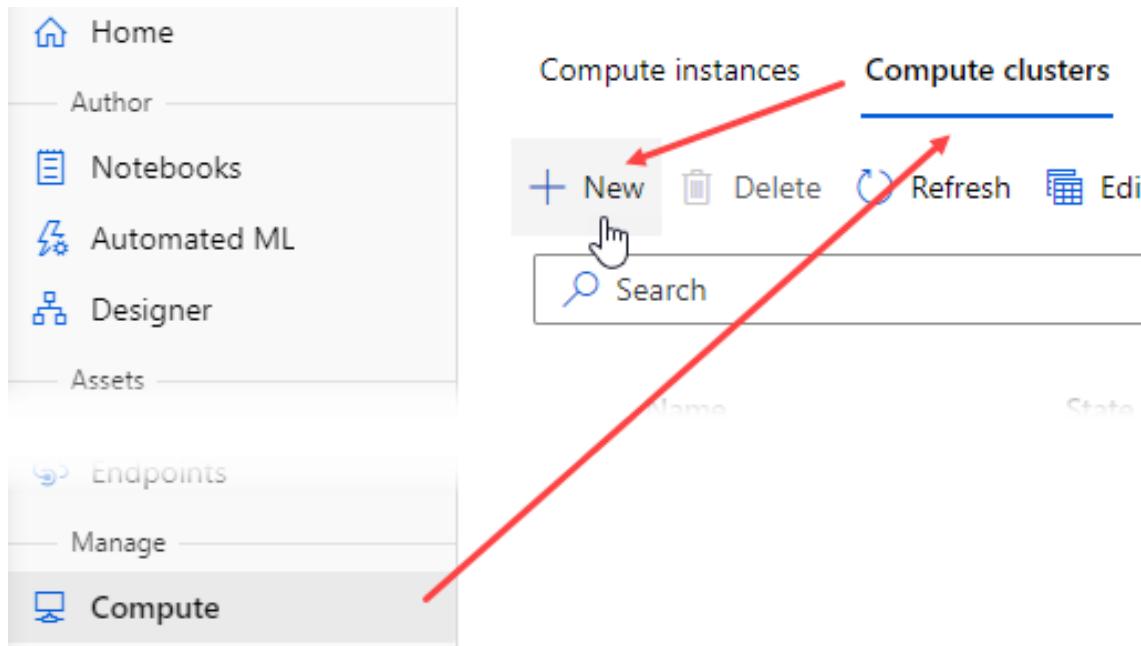


Figure 13.27 – Creating a new compute cluster

10. Then you can select your preferred location for the cluster and the virtual machine type and size to use for each cluster node. You can leave the default selection and click **Next**.
11. Choose a name for your cluster (in our case, `cluster`), the minimum number of nodes (keep it at 0 to make it turn off automatically when not used), and the maximum number of nodes (set it to 2). Then, click on **Create** to allocate your compute cluster.

12. Now, click on the **Automated ML** tab on the left menu and then on **New Automated ML run**.



Figure 13.28 – Creating a new AutoML experiment

Figure 13.28 – Creating a new AutoML experiment

13. On the next page, select the **titanic-imputed** dataset and click **Next**.

14. Now you can configure the run by entering the name of the new experiment (a virtual folder) that will contain all the AutoML runs (we used `titanic` for the name), the machine learning target column (`Survived`, the one to predict), and the compute cluster to use to execute the AutoML runs (the `cluster` one created previously).



Figure 13.29 – Configuring your AutoML run

Figure 13.29 – Configuring your AutoML run

15. You can then declare the machine learning experiment type you would like to run. In our case, it is a classification.



Figure 13.30 – Setting up the AutoML task type

Figure 13.30 – Setting up the AutoML task type

16. By clicking on **View additional configuration settings**, you can choose the primary metric to use in your experiment. Select the **AUC weighted** one and then click **Save**.
17. By clicking on **View featurization settings**, you can enable the auto-featurization option that AutoML provides. By default, it's switched on. You can also choose the feature type for each column and the missing values impute strategy for each of them (the strategies are the naïve ones). Keep everything on **Auto** and then click **Save**.
18. You can now click **Finish** to start your AutoML experiment. You'll be redirected to the **Run** page, and after a while, you'll see your experiment running.



Figure 13.31 – Your AutoML experiment running
Figure 13.31 – Your AutoML experiment running

19. After about 30 minutes, the experiment should end. Click on the **Models** tab on the **AutoML Run** page and you will see the training pipelines according to the best-performing ones.



Figure 13.32 – Best performing pipelines found by AutoML
Figure 13.32 – Best performing pipelines found by AutoML

20. For the best-performing model (**VotingEnsemble**), the *Explainability Dashboard* is also automatically generated, which you can access by clicking on **View explanation**. For further details on this, check out the references. Now, click on the **VotingEnsemble** link to go to the specific run that trained the model using that pipeline. Then, click on the **Deploy** button.



Figure 13.33 – Deploying the best model to a web service
Figure 13.33 – Deploying the best model to a web service

A new form will appear on the right asking for information about the model to deploy on a web service. Just give the model endpoint a name (`titanic-model`), select **Azure Container Instance** as the compute type, as this will not be a production environment, and activate the **Enable authentication** feature. In the case of a production environment, *Azure Kubernetes Services (AKS)* is the best choice. Then, click on **Deploy** and wait for the model to be deployed. When the **Deploy status** field changes to **Succeeded** in the **Model** summary, click on the **titanic-model** endpoint link.

21. The endpoint **Details** page contains all the information about the service. After at least 10 minutes of deployment, it must be in a healthy deployment state in order to be used. You can click on the **Test** tab to test your endpoint by providing it with test input data. The tab we are most interested in is **Consume**, in which the coordinates (REST

endpoint URL and authentication key) are indicated to invoke the REST API from an external system. Also, you can directly copy the code snippet that allows you to consume the service in Python in the **Consumption option** section. We will use a variation of this code to score test observations in Power Query.

At this point, the model is ready on a web service to be consumed via REST APIs. Let's now use it in Power Query.

Consuming an Azure ML deployed model in Power BI

Using a variation of the Python code proposed by the **Endpoint Consume** tab on Azure ML Studio, we created a function that accepts as parameters the endpoint URL, the API key, and a dataframe containing the observations to be scored. In the output, we get a dataframe containing just the `predicted_label` column with the scoring of each observation.

Here are the steps to get the predictions of a test dataset from a model trained via Azure AutoML and deployed as a web service on an Azure container instance:

1. Click on **Get Data**, select **Text/CSV**, and then click on **Connect**:
2. Select the `titanic-test.csv` file in the `Chapter13` folder and then click **Open**.
3. You'll see a preview of the test data. Click **Transform Data**.
4. Click **Transform** on the ribbon and then **Run Python script**.
5. Enter the script you can find in the
`06-use-azure-ml-web-service-in-power-bi.py` file in the
Chapter13\Python folder. Remember to edit the endpoint URL and
key accordingly.
6. We are only interested in the `scored_df` dataframe. So, click on its
Table value.
7. You'll see a preview of the test dataset with an additional column –
`predicted_label`.
8. Click **Home** on the ribbon and then click **Close & Apply**.

Amazing! You were able to consume a model trained on Azure Machine Learning and deployed it to an Azure container instance without having either a PPU license or a Premium capacity.

Using cognitive services in Power Query

The Azure cognitive services **Text Analytics API** is a service that provides Natural Language Processing (NLP) functions for text mining and analysis. Features made available include sentiment analysis, opinion mining, key phrase extraction, language detection, and named entity recognition.

First, you need to deploy the text analytics resource via the Azure portal.

Configuring text analytics

You must have an Azure subscription to use these services. Then you need to create a text analytics resource by following these steps:

1. Go to the Azure portal (<https://portal.azure.com/>) and click on the **Create a resource** plus icon.
2. Start entering the text string in the search textbox and the **Text Analytics** option will appear. Click on it. Then, click on the **Create** button on the **Text Analytics** page.
3. Forget about selecting the **Custom question answering** option, and click on **Continue to create your resource** instead.
4. On the **Create Text Analytics** page, select the region you prefer and give a name to the service (in our case, `textanalytics555`; you can use a unique name of your choosing). Assign your resource to a new resource group with the name `text-analytics`. Then, select the **Free F0** pricing tier, check the **Responsible AI Notice** option, and click on **Review + create**. Then, click **Create** on the next page.
5. Once the deployment of the resource is complete, click on **Go to resource** and click on the **API Key** link on the next page. Then, take note of the details of **KEY 1** (you can click on the **Copy to clipboard** icon on its right) and the endpoint URL. You'll use this information in your Python code.

Your resource is now ready to be used via the dedicated Python SDK.

Configuring your Python environment and Windows

In order to consume text analytics, you must first install the **Microsoft Azure Text Analytics Client Library for Python** by following these steps:

1. Open your Anaconda Prompt.
2. Switch to your PyCaret environment by entering this command:
`conda activate pycaret_env .`
3. Install the client library by entering this command:
`pip install azure-ai-textanalytics==5.1.0 .`

After that, to avoid the *ssl module in Python is not available* error in Windows 10, you need to add the `pycaret_env\Library\bin` path to the Windows environment PATH variable. These are the steps to do it:

1. Click on the Windows **Start** icon in the bottom-left corner of your screen and start digitizing the `x` variable `.string`. This will search for all Windows options that have the string variable in their name. Then, click on **Edit environment variables for your account**.
2. In the **Environment Variables** windows, double-click on the **Path** variable under **User variables for <your-user>** (if you installed Miniconda for all users, you need to change the **Path** system variable). In the **Edit environment variable** dialog that will appear, click on the **New** button and add the path
`C:\<your-path>\miniconda3\envs\pycaret_env\Library\bin .`
Then, click **OK** on all the windows.
3. You need to restart your system to make the change effective.

You are now ready to be able to consume the service from Power Query.

Consuming the Text Analytics API in Power BI

In this section, we will show you how to do sentiment analysis thanks to text analytics on the fictional company *Fabrikam Fiber*. It provides cable television and related services in the United States, allowing users to enter

comments on their website. Your goal is to define for each comment the degree of positivity, neutrality, and negativity.

Basically, once a client has been authenticated with a URL and key, you can easily carry out a sentiment analysis thanks to the `analyze_sentiment()` method without knowing any NLP basis. Keep in mind that the free tier of text analytics is limited to processing only 10 documents (in our case, comments) at a time. For this reason, the code we built consists of grouping the comments in groups of 10 and invoking the API for each group.

Let's see how to do that:

1. Click on **Get Data**, select **Text/CSV**, and then click on **Connect**:
2. Select the `FabrikamComments.csv` file in the `Chapter13` folder and click **Open**.
3. You'll see a preview of the Fabrikam dataset. Then, click **Transform Data**.
4. Click **Transform** on the ribbon, followed by **Run Python script**.
5. Enter the script you can find in the `07-use-text-analytics-in-power-bi.py` file in the `Chapter13\Python` folder. Remember to appropriately replace the service URL and its key that you previously copied into the Azure portal.
6. We are only interested in the `sentiment_enriched_df` dataframe. So, click on its **Table** value.
7. You'll see a preview of the Fabrikam dataset enriched with the following additional columns: `comment_sentiment`, `overall_positive_score`, `overall_neutral_score`, and `overall_negative_score`:

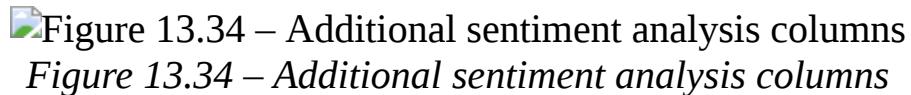


Figure 13.34 – Additional sentiment analysis columns
Figure 13.34 – Additional sentiment analysis columns

8. Click **Home** on the ribbon and then click **Close & Apply**.

That's amazing! Thanks to the Python library `azure.ai.textanalytics`, you were able, in a few lines of code, to perform sentiment analysis in a very simple way. With the same ease, you can also use in Power BI the other services that cognitive services provide thanks to other Python SDKs.

Summary

In this chapter, you learned how Power BI interacts with Microsoft AI services by default through data flow features. You also learned that by using AutoML platforms, you can get around the licensing problem (PPU license or Premium capacity) that Power BI needs to interface with Microsoft AI services. You used both an on-premises AutoML solution (PyCaret) and Azure AutoML on the cloud to solve a binary classification problem. You also used cognitive services' text analytics to do some sentiment analysis directly using a Python SDK.

You've learned that enrichment via AI mostly happens in Power Query (which allows access to the internet), although you've seen a case where it may be convenient to use a machine learning model directly within a Python visual.

In the next chapter, you will see how to implement data exploration of your dataset in Power BI.

References

For additional reading, check out the following books and articles:

1. *AI with data flows* (<https://docs.microsoft.com/en-us/power-bi/transform-model/dataflows/dataflows-machine-learning-integration>)
2. *A Review of Azure Automated Machine Learning (AutoML)* (<https://medium.com/microsoftazure/a-review-of-azure-automated-machine-learning-automl-5d2f98512406>)
3. *Automated Machine Learning with Microsoft Azure*, by Dennis Michael Sawyers, Packt Publishing

(<https://www.amazon.com/Automated-Machine-Learning-Microsoft-Azure/dp/1800565313/>)

4. A Gentle Introduction to Concept Drift in Machine Learning (<https://machinelearningmastery.com/gentle-introduction-concept-drift-machine-learning/>)
5. Machine Learning Basics with the K-Nearest Neighbors Algorithm (<https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761>)
6. Python's «*predict_proba*» Doesn't Actually Predict Probabilities (and How to Fix It) (<https://towardsdatascience.com/pythons-predict-proba-doesn-t-actually-predict-probabilities-and-how-to-fix-it-f582c21d63fc>)
7. Use the Interpretability Package to Explain ML Models and Predictions in Python (<https://docs.microsoft.com/en-us/azure/machine-learning/how-to-machine-learning-interpretability-aml>)
8. Get Started with Tidymodels (<https://www.tidymodels.org/start/> <https://www.tidymodels.org/start/>)

14 Exploratory Data Analysis

In *Chapter 13, Using Machine Learning without Premium or Embedded Capacity*, we mentioned that using **Auto Machine Learning (AutoML)** solutions on a dataset blindly often does not lead to very accurate models. This is because it is necessary to understand the most inherent characteristics of the dataset by using statistical tools at an earlier stage to extract useful information in order to get a better model.

The approach to be used for this type of dataset analysis is called **Exploratory Data Analysis (EDA)** and was first introduced by John Turkey to encourage statisticians to explore data and formulate hypotheses that would lead to new data collection and experiments to eventually enrich patterns among the variables in a dataset.

In this chapter, you will learn about the following topics:

- What is the goal of EDA?
- Exploratory Data Analysis with Python and R
- Exploratory Data Analysis in Power BI

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

What is the goal of EDA?

The objective of EDA is to make sure that the dataset to be used later for more complex processes is first of all clean, that is, it has no missing values and no outliers that could divert possible subsequent analyses. In addition, it

is important to select during this phase the variables that actually bring information, trying to drop those that determine mostly noise. This eliminates possible sources of inaccuracy in the conclusions to which subsequent processes lead. At this stage, it is also important to study the associations between variables and gain insights from the data analyzed in order to justify any more complex processing to be applied later.

Ultimately, the phases of EDA are as follows:

1. Understanding your data
2. Cleaning your data
3. Discovering associations between variables

Let's look in detail at what types of analysis they involve.

Understanding your data

In this first phase, it is essential to understand the meaning that each variable takes on in the context of the problem that the dataset represents. Once the measurable business entities with which the variables are associated are clear, it is easier to draw conclusions about how they interact with each other.

Having an idea of **the size of the dataset**, understood as the number of variables and the number of observations (rows), helps you get a first idea of the size of the data you will be dealing with. After that, identifying and immediately defining the **type of variables** involved (which can be numerical or categorical) is of crucial importance in order to visualize them in the most appropriate way.

Then, knowing the **descriptive statistics** of the **numerical variables** in the dataset helps to have greater sensitivity about their values. For each numeric variable, you can represent its histogram by highlighting its skewness and boxplot to better analyze its distribution, as you have already studied in *Chapter 11, Adding Statistics Insights: Associations*.

For **categorical variables**, on the other hand, it is important to know the number of distinct categories and which characteristics of the observations

they identify. It is possible to represent each unique category with its frequency and relative percentage of the total in a **bar chart**. Here is an example considering the Titanic disaster dataset:

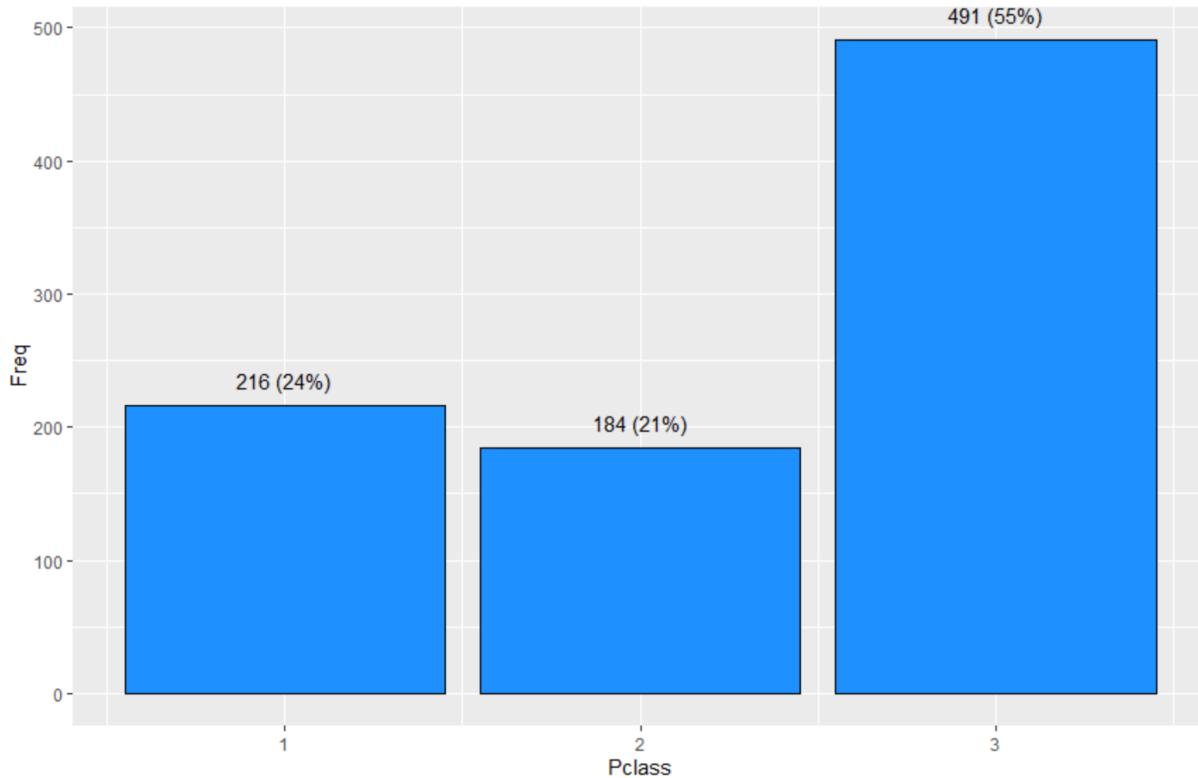


Figure 14.1 – Bar chart of Pclass

It is important to check whether more than one category has a very low frequency because it might make sense to aggregate them into a single category. This is usually done in a subsequent phase of a machine learning project (called **Feature Engineering**) in order to avoid, for example, having a model that is not generic enough for all the cases for which it must be adopted (the phenomenon of **overfitting**) and therefore determining bad performance.

It is also very important to give the analyst the possibility of graphically representing a pair of variables in order to understand how they interact with each other (**multivariate analysis** of the dataset). The type of visualization obviously changes according to the data type of the variables involved: in the case of two numeric variables, you will have a scatter plot;

in the case of two categorical variables, you will have a mosaic plot; in the case of one numeric and one categorical variable, you will have several boxplots (or raincloud plots) for each category.

At this point, we can move on to describing the phase that deals with data cleaning.

Cleaning your data

As you learned in *Chapter 12, Adding Statistics Insights: Outliers and Missing Values*, the most important activities to obtain a statistically robust dataset that is as close as possible to the real case it represents are these two:

- Determining **outliers** and managing them
- Imputing **missing values**

During the EDA process, it is important to make use of all visual and non-visual tools useful to detect outliers and missing values in a straightforward way. This will allow any anomalies found to be fixed at a later stage.

As you have learned previously, for **univariate outlier analysis** it is very useful to represent individual distributions using boxplots for numeric variables. And it would also be very useful to be able to see what outliers remain or are added after transforming the variable via Yeo-Johnson transformations. That way you can see if it might be worth trying to normalize the distribution to deal with as few outliers as possible.

It is also convenient to be able to do a **bivariate analysis of the outliers** considering a numerical variable and a categorical one so that you study the anomalies of the distribution of the numerical variable broken down by each label of the categorical one.

After carefully studying and identifying possible outliers, it is also important to have a convenient visualization of combinations and intersections of **missing values** between variables. The goal is therefore to know not only the count and relative percentage value of the missing values

of each variable, but also to have a convenient visualization of how these missing values interact with each other thanks to an upset plot.

Finally, let's turn to the analysis of associations between variables.

Discovering associations between variables

As you learned in *Chapter 11, Adding Statistics Insights: Associations*, knowing the **degree of association between variables** in the dataset certainly helps you understand which ones carry the most information and which ones generate mostly just noise. Selecting the most informative variables and using them for training a machine learning model certainly helps you to get more stable and performant results.

Having visual tools to interpret the associations between pairs of numeric-numeric, numeric-categorical, and categorical-categorical variables certainly helps in the hard task of feature selection.

There are tools that allow you to generate EDA reports automatically. Let's see what they are.

Exploratory Data Analysis with Python and R

If you have to do data exploration using only Python or R, there are tools that automatically generate a series of visuals that make your life easier. We have two lists below, one for Python tools, the other for R tools, in case you need any. It is easier to find tools for interactive data analysis in Python than in R. Packages available in R often provide wrappers that greatly simplify EDA via coding.

The Python libraries for EDA are the following:

- **SweetVIZ** (<https://pypi.org/project/sweetviz/>): An open source Python library that generates beautiful, high-density visualizations to kickstart EDA with just two lines of code.
- **Lux** (<https://lux-api.readthedocs.io/>): A Python library that facilitates fast and easy data exploration by automating the visualization and data

analysis process.

- **Pandas Profiling** (<https://pandas-profiling.github.io/pandas-profiling/>): Generates profile reports from a pandas dataframe for data analysis.
- **PandasGUI** (<https://github.com/adamerose/pandasgui>): A GUI for viewing, plotting, and analyzing pandas dataframes.
- **D-Tale** (<https://github.com/man-group/dtale>): The combination of a Flask backend and a React frontend to bring you an easy way to view and analyze pandas data structures.

The R packages for EDA are the following:

- **ExPanDaR** (<https://github.com/joachim-gassen/ExPanDaR>): ExPanD is a Shiny-based app supporting interactive exploratory data analysis.
- **ggquicKeda** (<https://github.com/smouksassi/ggquicKeda>): An R Shiny app/package that enables you to quickly explore your data and to detect trends on the fly thanks to scatterplots, dot plots, boxplots, bar plots, and so on.
- **summarytools** (<https://github.com/dcomtois/summarytools>): An R package for data cleaning, exploring, and simple reporting. It integrates well with commonly used software and tools for reporting in R (such as RStudio).
- **DataExplorer** (<https://boxuancui.github.io/DataExplorer/>): Aims to automate most data handling and visualization so that users can focus on studying the data and extracting insights.

Take a look at the references for further details about the aforementioned tools.

If you want to take advantage of Power BI as a visualization tool, you unfortunately cannot use many of the visualizations automatically generated by the aforementioned Python tools. Therefore, we will use R packages to generate the necessary for basic EDA via code because R allows you to obtain very high-quality graphics of any complexity quite easily. Let's see how.

Exploratory data analysis in Power BI

In this section, we will make extensive use of the `GGplot2` package, an advanced R library designed for creating plots based on **The Grammar of Graphics**. It is not our intention to go into detail about every feature exposed by the package, even if it is used quite widely in the code that accompanies the chapter. Our goal is, as always, to provide code that is easy to customize for use in other projects, and above all, to provide a starting point for a more detailed look at the functions used. For more details, take a look at the references of this chapter.

In addition to the tools that Tidyverse provides in R (including `ggplot2`), we will also use the `summarytools` and `DataExplorer` packages to create EDA reports in Power BI. It is therefore necessary to install them:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command:
`install.packages('Rcpp')`. It's a `summarytools` dependency.
Then, press *Enter*.
3. Enter this command: `install.packages('summarytools')`. Then, press *Enter*.
4. Enter this command: `install.packages('DataExplorer')`. Then, press *Enter*.

At this point, we can start implementing our EDA reports. First, however, it is important to highlight the fact that the data type of variables must be defined in each R script you will use in your report.

Important Note

For the purpose of EDA, it is important to define the data type of variables in the dataset to be analyzed. Depending on the data types of the features, different analysis strategies are applied. The data types of columns of a Power BI table in the data model are not mappable one-to-one with the data types handled by R. Therefore, it is necessary to define the data types of the columns directly in an R script to prevent Power BI from overwriting them.

Moreover, the `00-init-dataset.R` script that you will find in the `Chapter14\R` folder will be used as the *official CSV connector* of your EDA report in Power BI as in this case it is more convenient to load data directly into R for each script in which it is used. After loading your data in it, you need to define the columns to drop, the categorical and integer ones. Columns specified as integers will be viewed as either numeric or categorical variables, leaving the analyst to select them according to the function that best describes the phenomenon under analysis. All the other columns are supposed to be numeric:

```
library(readr)
library(dplyr)
dataset_url <- 'http://bit.ly/titanic-dataset-csv'
src_tbl <- read_csv(dataset_url)
vars_to_drop <- c('PassengerId')
categorical_vars <- c('Sex', 'Ticket', 'Cabin', 'Embarked')
integer_vars <- c('Survived', 'Pclass', 'SibSp', 'Parch')
```

If you need to import an Excel file instead of a CSV file, you already know how to implement changes to the code because of what you learned in *Chapter 7, Logging Data from Power BI to External Sources*.

Then the script will define the `tbl` tibble applying the appropriate transformations:

```
tbl <- src_tbl %>%
  mutate(
    across(categorical_vars, as.factor),
    across(integer_vars, as.integer)
  ) %>%
  select( -all_of(vars_to_drop) )
```

This script will be pre-loaded into every other R script you develop for the EDA report so that you are sure to have the dataset with the precise data types specified ready in memory.

So, let's start developing the report from the basic statistics of the dataset.

Dataset summary page

The first page of the EDA report is responsible for providing the analyst with an overview of the data contained in the dataset. Some **basic information** will be exposed enriching the output of the function `introduce()` of the `DataExplorer` package. This information is as follows:

- Number of rows and columns
- Number of discrete (categorical) and continuous (numeric) columns
- Number of columns having all missing values
- Total number of missing values in the dataset
- Number of complete rows, that is, not having missing values
- Total number of observations in the dataset, given by the number of rows multiplied by the number of columns
- Memory used in KB
- Number of duplicated rows

Then a **summary statistics table** for the entire dataset is calculated starting from the output of the `dfSummary()` function of the `summarytools` package. The information it contains for each variable is as follows:

- Variable name and data type
- Basic statistics according to the data type
- Number of unique valid values (not null)
- Frequencies of valid values
- Number of valid values and their percentage
- Number of missing values and their percentage

Finally, more extensive **descriptive statistics** are provided for numerical variables thanks to the `descr()` function of the `summarytools` package. In addition to the most common ones, the following statistics are added:

- **Median Absolute Deviation (mad):** Contrary to the variance and standard deviation, it is a robust measure of outliers of how widespread a dataset is. It is usually used for non-normal distributions.
- **Interquartile range (iqr):** As you already saw for the boxplots, it is a measure of where the bulk of the distribution values lie.
- **Coefficient of variation (cv):** It is a measure of relative variability as it is the ratio of the standard deviation to the mean.

- **Coefficient of skewness** (*skewness*): It is a measure of the lack of distribution symmetry.
- **Kurtosis** (*kurtosis*): It is a measure of whether the data is heavy-tailed or light-tailed relative to a normal distribution.

With this information, the analyst can get a complete picture of the shape of the data contained in the dataset.

Let's see how to implement them in Power BI. First, make sure that Power BI Desktop references the correct versions of R in **Options**. Then follow these steps:

1. Click on **Get Data**, search for `script`, select **R script**, and click on **Connect**.
2. Enter the script you can find in the file `01-basic-info.R` into the `Chapter14\R` folder. Make sure that the `init_path` variable points to the correct absolute path of the `00-init-dataset.R` file you can find in the same folder.
3. In the navigator dialog, select the `basic_info_tbl`, the `numeric_vars_descr_stats_tbl`, the `summary_tbl`, and the `sample_tbl` tables. Then click on **Load**.
4. After the four tables are loaded in the **Fields** panel on the right in Power BI Desktop, you can add them into your report canvas simply by checking their measures. For example, expand the `basic_info_tbl` content and check the fields `attribute` (string field) and `value` (numeric field) in this order to get a table chart (if you click on the numeric field first, you'll get a column chart). A table will appear on your canvas:

attribute	value
all_missing_columns	0.00
columns	11.00
complete_rows	183.00
continuous_columns	6.00
discrete_columns	5.00
duplicated_rows	0.00
memory_usage	184,456.00
rows	891.00
total_missing_values	866.00
total_observations	9,801.00
Total	196,219.00

Figure 14.2 – Basic info table in its first state

In this case, the **Total** row makes no sense. Therefore, it must be eliminated working on the **Format** properties of the visual.

5. Make sure that the table visual is selected and then click on the **Format** options (the wall brush icon) on the **Visualizations** panel on the right. Expand the **Total** section and switch off the **Total** option. Then expand the **Field formatting** section, select the **value** field

from the combo box, and enter `0` into the **Value decimal places** textbox. Finally, switch on the **Title** option and expand its options. Then enter the label `Basic Dataset Info` in the textbox.

6. Now click on an empty spot on the canvas, expand the `summary_tbl` fields and check them in the following order: **Variable**, **Stats Values**, **Unique Valid**, **Freq of Valid**, **Valid**, and **Missing**. After that, expand the right edge of the visual in order to see all the columns.
7. As done in *Step 5* for the `basic_info_tbl` table, remove the **Total** row also for this table and use this string as the label:
`Dataset Summary`.
8. Now click on an empty spot on the canvas, expand the `numeric_vars_descr_stats_tbl` fields, and check them in the following order: **variable**, **mean**, **sd**, **min**, **q1**, **med**, **q3**, **max**, **mad**, **iqr**, **cv**, **skewness**, and **kurtosis**. After that, expand the right edge of the visual in order to see all the columns.
9. For this table, also remove the **Total** row and use this string as the label: `Descriptive Statistics for Numeric Variables`.
10. Now click on an empty spot on the canvas, expand the `sample_tbl` fields, and check them all in the order you want after checking **Name** first (checking a non-numeric field at the beginning, the visual selected by default will be the table).
11. For this table, also remove the **Total** row and use this string as the label: `Dataset Sample`.
12. After repositioning the visuals on the canvas and expanding the individual table columns a bit, you'll get your first EDA report page, like the following:

Dataset Summary							Descriptive Statistics for Numeric Variables												
Variable	Stats Values	Unique Valid	Freq of Valid	Valid	Missing		variable	mean	sd	min	q1	med	q3	max	mad	iqr	cv	skewness	kurtosis
Age [numeric]	Mean (sd) : 29.7 (14.5) min < med < max: 0.4 < 28 < 80 IQR (CV) : 17.9 (0.5)	88 distinct values	88 distinct values	714 (80.1%)	177 (19.9%)		Fare	32.20	49.69	0.00	7.90	14.45	31.00	512.33	10.24	23.09	1.54	4.77	33.12
Cabin [factor]	1. A10 2. A14 3. A16 4. A19 5. A20 6. A23 7. A24 8. A26 9. A31	147 distinct values	1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%) 1 (0.5%)	204 (22.9%)	687 (77.1%)		Age	29.70	14.53	0.42	20.00	28.00	38.00	80.00	13.34	17.88	0.49	0.39	0.16
							SibSp	0.52	1.10	0.00	0.00	0.00	1.00	8.00	0.00	1.00	2.11	3.68	17.73
							Parch	0.38	0.81	0.00	0.00	0.00	0.00	6.00	0.00	0.00	2.11	2.74	9.69
							Pclass	2.31	0.84	1.00	2.00	3.00	3.00	3.00	0.00	1.00	0.36	-0.63	-1.28
							Survived	0.38	0.49	0.00	0.00	0.00	1.00	1.00	0.00	1.00	1.27	0.48	-1.77

Basic Dataset Info		Dataset Sample															
attribute	value	Name	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived					
all_missing_columns	0	Ahllin, Mrs. Johan (Johanna Persdotter Larsson)	3	female	40.00	1	0	7546	9.48		S	0					
columns	11	Allen, Mr. William Henry	3	male	35.00	0	0	373450	8.05		S	0					
complete_rows	183	Andersson, Miss. Erna Alexandra	3	female	17.00	4	2	3101281	7.93		S	1					
continuous_columns	6	Andersson, Mr. Anders Johan	3	male	39.00	1	5	347082	31.28		S	0					
discrete_columns	5	Andreasson, Mr. Paul Edvin	3	male	20.00	0	0	347466	7.85		S	0					
duplicated_rows	0	Arnold-Franchi, Mrs. Josef (Josefine Franchi)	3	female	18.00	1	0	349237	17.80		S	0					
memory_usage	184,456	Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)	3	female	38.00	1	5	347077	31.39		S	1					
rows	891	Backstrom, Mrs. Karl Alfred (Maria Mathilda Gustafsson)	3	female	33.00	3	0	3101278	15.85		S	1					
total_missing_values	866	Beesley, Mr. Lawrence	2	male	34.00	0	0	248698	13.00	D56	S	1					
total_observations	9,801	Bing, Mr. Lee	3	male	32.00	0	0	1601	56.50		S	1					
		Bonnell, Miss. Elizabeth	1	female	58.00	0	0	113783	26.55	C103	S	1					
		Braund, Mr. Owen Harris	3	male	22.00	1	0	A/5.21171	7.25		S	0					
		Caldwell, Master. Alden Gates	2	male	0.83	0	2	248738	29.00		S	1					
		Cann, Mr. Ernest Charles	3	male	21.00	0	0	A/5.2152	8.05		S	0					
		Carrau, Mr. Francisco M	1	male	28.00	0	0	113059	47.10		S	0					
		Celotti, Mr. Francesco	3	male	24.00	0	0	343275	8.05		S	0					
		Chaffee, Mr. Herbert Fuller	1	male	46.00	1	0	W.E.P. 5734	61.18	E31	S	0					

Figure 14.3 – Summary page of the EDA report

13. Just rename the page (right-click on the bottom page tab and click **Rename Page**) using the label **Summary**.

Keep the following in mind:

Important Note

In order to display the sample details of the dataset, you must properly select its fields in Power BI Desktop. This means that if you want to change the source dataset, you must edit the sample details visual by hand.

Nice work! You just developed the first page of your EDA report.

Let's now try to develop a page dedicated to the analysis of missing values.

Missing values exploration

Although some information about the missing values is present in text format on the newly developed **Summary** page, providing a page

containing visuals dedicated to the missing values is definitely an advantage for the analyst.

In addition to displaying a simple **lollipop plot** indicating the percentage values of missing values for each variable, you will also use an **upset plot** to allow the analyst to figure out which combinations of variables are found to be missing altogether, just as you learned in *Chapter 12, Adding Statistics Insights: Outliers and Missing Values*.

So, let's start to develop the **Associations** page:

1. If it does not already exist, create a new Power BI report page by clicking on the plus icon near the tabs at the bottom of the canvas. A new tab named **Page 1** will be created.
2. Now click on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane and enable it when you're prompted to do so. After that, move it to the left of the page and resize it in order to fill half of it.
3. Keeping it selected, click on the attribute field of the `basic_info_tb1` table. You won't use this field in the R Visual code, but you must select at least one measure to enable the script in the visual.
4. Now click on the R Visual's **Format** tab and under the **Title** section, enter the string `Missing Value Percentages` as the title.
5. Copy the code of the file `02-missing-values-plot-1.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top right of the R script editor (enable the R Visual every time it's requested). You'll get the lollipop plot in it.
6. Click again on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane, and enable it when you're prompted to do so. After that, move it to the right of the page and resize it in order to fill half of it.
7. Keeping it selected, click on the attribute field of the `basic_info_tb1` table. You won't use this field in the R Visual code, but you must select at least one measure to enable the script in the visual.

8. Now click on the R Visual's **Format** tab and under the **Title** section, enter the string **Variables Missing Together** as the title.
9. Copy the code of the file `03-missing-values-plot-2.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top right of the R script editor (enable the R Visual every time it's requested). You'll get the upset plot in it.
10. Just rename the current page (right-click on the tab at the bottom of the page and click **Rename Page**) using the label **Missing Values Analysis**.

Nice one! It is now much easier for the analyst to understand the impact of missing values on the dataset. The whole page will be like the following one:

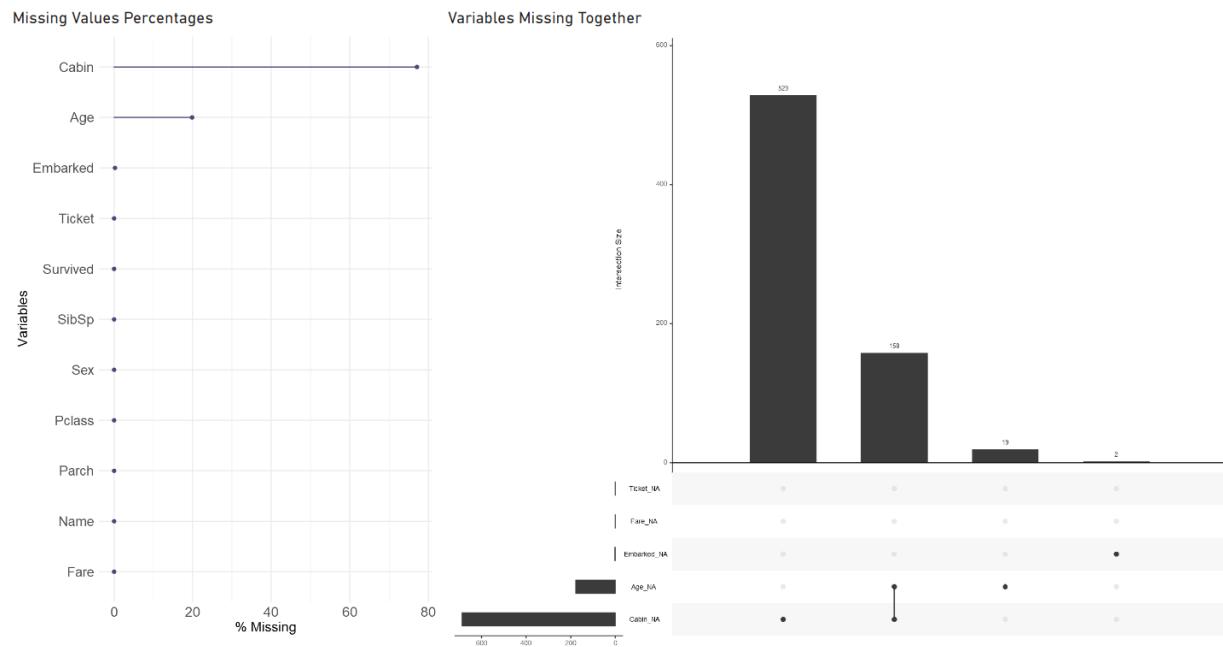


Figure 14.4 – Missing values analysis page of the EDA report

Let's now turn to developing a page dedicated to univariate analysis of the dataset.

Univariate exploration

The one-dimensional analysis of a dataset consists of the analysis of the distributions of the variables that compose it. Depending on the data type of the variables, the visual tools used to facilitate this analysis are different.

For the analysis of numerical variables, a couple of graphs are used to better delineate their distribution: there is a **histogram and density plot** overlapping in the first graph, while in the second graph there is a **raincloud plot** (the set of a density plot and a boxplot). In *Figure 14.5*, you can find an example of the graphs for a numeric variable:

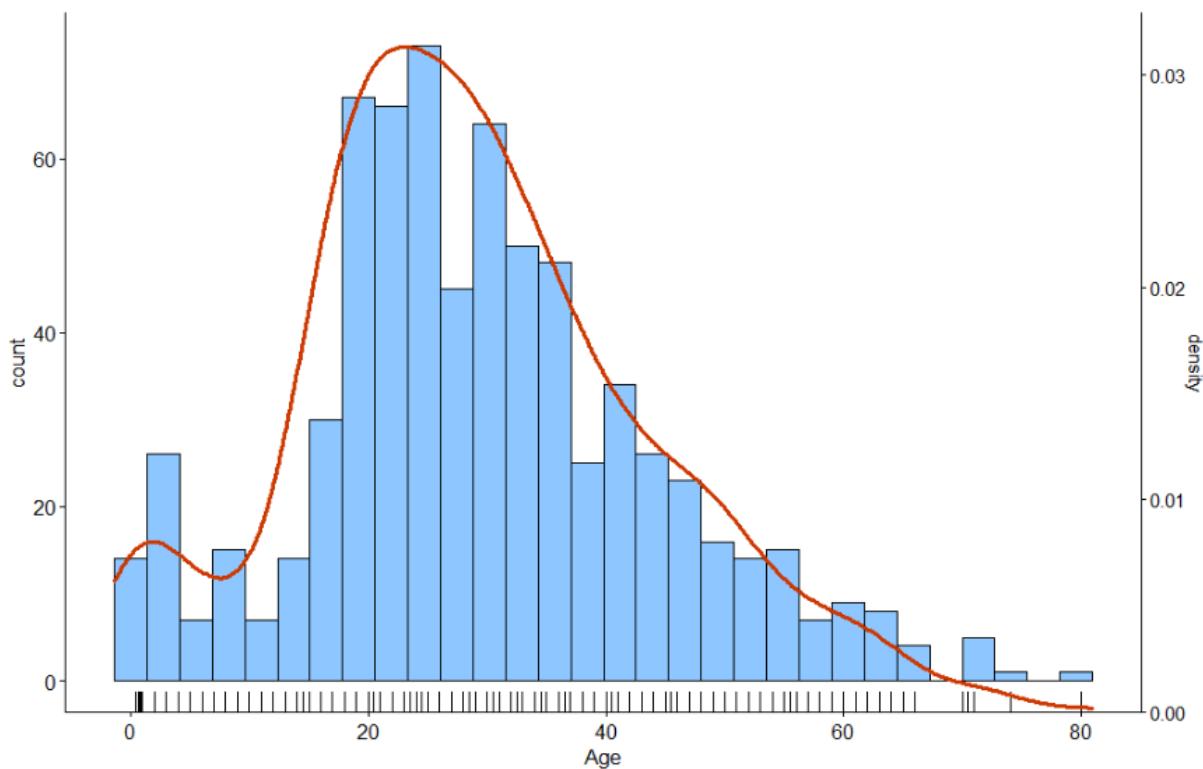


Figure 14.5 – Histogram and density plot of Age

For categorical variables, a simple **bar chart** is used, with the intention of limiting the maximum number of distinct categories to be displayed so that they don't exceed a predefined number, and grouping the other categories in the dummy **others** category:

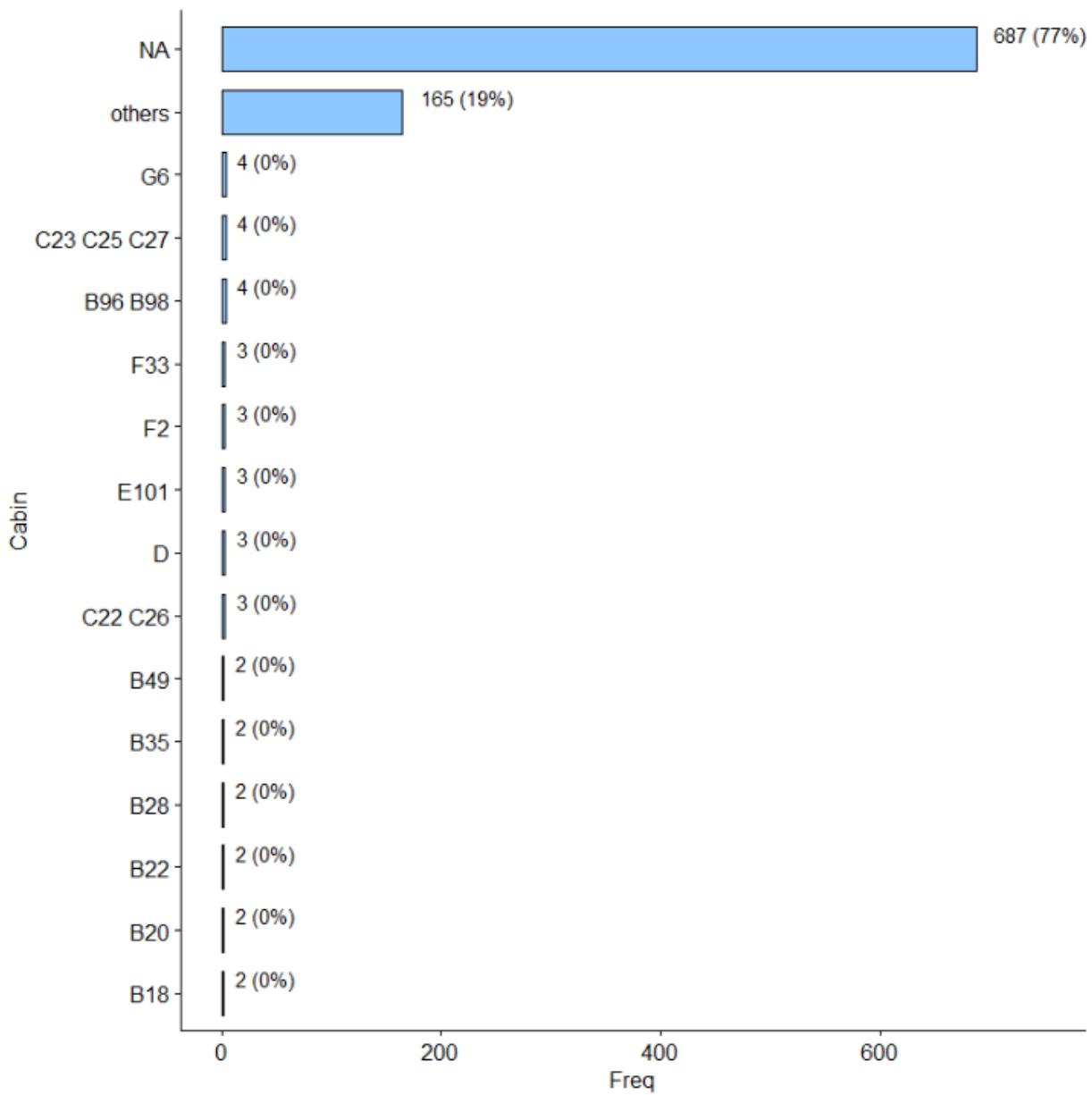


Figure 14.6 – Bar chart of the Cabin variable

The selection of variables to be displayed in Power BI is done through drop-down slicers. If a variable is of the integer type, it appears both in the dropdown of the numerical variables and in that of the categorical variables, so the analyst can visualize them in the way that they consider more appropriate.

Since the graphs will need to be displayed via R Visuals, should you wish to publish the EDA report to the Power BI service, you must ensure that the R

packages to be used are compatible with the version of the R engine found on the Power BI service. Remember also that Power BI Desktop can reference only one R engine at a time, which will be the one used for both Power Query and R Visuals on your machine. So, if you plan to use the latest R engine for Power Query once the report is published, you should also install the packages needed to render the report on this engine in your machine. This way you ensure that the report is rendered correctly on Power BI Desktop as well during your tests:

1. Open RStudio and make sure it is referencing your CRAN R dedicated to R Visuals (version 3.4.4 in our case).
2. Click on the **Console** window and enter this command:
`install.packages('cowplot')`. It's like an extension of `ggplot2`, providing themes and functions to align, arrange, and annotate plots. Then, press *Enter*.
3. Enter this command: `install.packages('ggpubr')`. It provides some easy-to-use functions for creating and customizing `ggplot2` based publication-ready plots. Then, press *Enter*.
4. Enter this command: `install.packages('ggExtra')`. It is a collection of functions and layers to enhance `ggplot2`, mostly used for its `ggMarginal` function, which adds marginal histograms/boxplots/density plots to scatterplots. Then, press *Enter*.
5. Enter this command: `install.packages('RColorBrewer')`. It is an essential tool to manage colors and palettes with R. Then, press *Enter*.
6. Enter this command: `install.packages('ggmosaic')`. It is designed to create visualizations of categorical data, in particular mosaic plots. Then, press *Enter*.
7. Enter this command: `install.packages(' ggdist')`. It is designed to create visualizations of categorical data, in particular mosaic plots. Then, press *Enter*.
8. Change the RStudio R reference to your CRAN R dedicated to Power Query (version 4.0.2 in our case) and restart RStudio.
9. Repeat steps 2 to 7.

Now you can start developing the new page of your report dedicated to the univariate analysis of the dataset:

1. If it does not already exist, create a new Power BI report page by clicking on the plus icon near the tabs at the bottom of the canvas. A new tab named **Page 1** will be created.
2. Click on **Get Data**, search for `script`, select **R script**, and click on **Connect**.
3. Enter the script you can find in the file `04-serialize-univariate-plots.R` into the `Chapter14\R` folder. Make sure that the `folder` variable points to the correct absolute path of the `Chapter14` folder.
4. In the navigator dialog, select the `basic_info_tbl`, the `numeric_vars_descr_stats_tbl`, the `summary_tbl`, and the `sample_tbl` tables. Then click on **Load**.
5. This script serializes to disk the lists with the graphs associated with each variable according to their type. Once you click **OK**, you will find the following files on disk: `histodensity_1st.rds`, `histodensity_transf_1st.rds` (numerical variables transformed according to Yeo-Johnson), and `barchart_1st.rds`. Pre-computing lists of graphs and serializing them to disk is a good strategy when the possible combinations of the variables involved are low.
6. In the Navigator dialog, select the `categorical_df` and `numeric_df` tables. As you can see, the `numeric_df` table contains the names of the numeric variables in the `numeric_col_name` column repeated twice, once for each type of transformation in the `transf_type` column to be applied to them (`standard` and `yeo-johnson`):

numeric_df

numeric_col_name	transf_type
Survived	standard
Pclass	standard
Age	standard
SibSp	standard
Parch	standard
Fare	standard
Survived	yeo-johnson
Pclass	yeo-johnson
Age	yeo-johnson
SibSp	yeo-johnson
Parch	yeo-johnson
Fare	yeo-johnson

Figure 14.7 – numeric_df table's content

Then click on **Load**.

7. Click on the slicer visual icon. Then expand the `numeric_df` table under the **Fields** panel and check the `numeric_col_name` measure.
8. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
9. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, switch off the **Slicer header** one, and add the **Title** Numeric Variables .
10. Click on an empty spot in the report canvas first. Then click on the slicer visual icon. Then expand the `numeric_df` table under the **Fields** panel and check the `transf_type` measure.
11. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
12. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, switch off the **Slicer header** one, and add the **Title** Transformations .
13. Now click on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane and enable it when you're prompted to do so. After that, move it under the two slicers you've just created.
14. Keeping it selected, click on both fields of the **numeric_df** table (`numeric_col_name` and `transf_type`).
15. Now click on the R Visual's **Format** tab and switch off the **Title** option.
16. Copy the code of the file `05-plot-numeric-variables.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top right of the R script editor (enable the R Visual every time it's requested). You'll get the univariate analysis of the not transformed `Age` variable:

Numeric Variables

Age

Transformations

standard

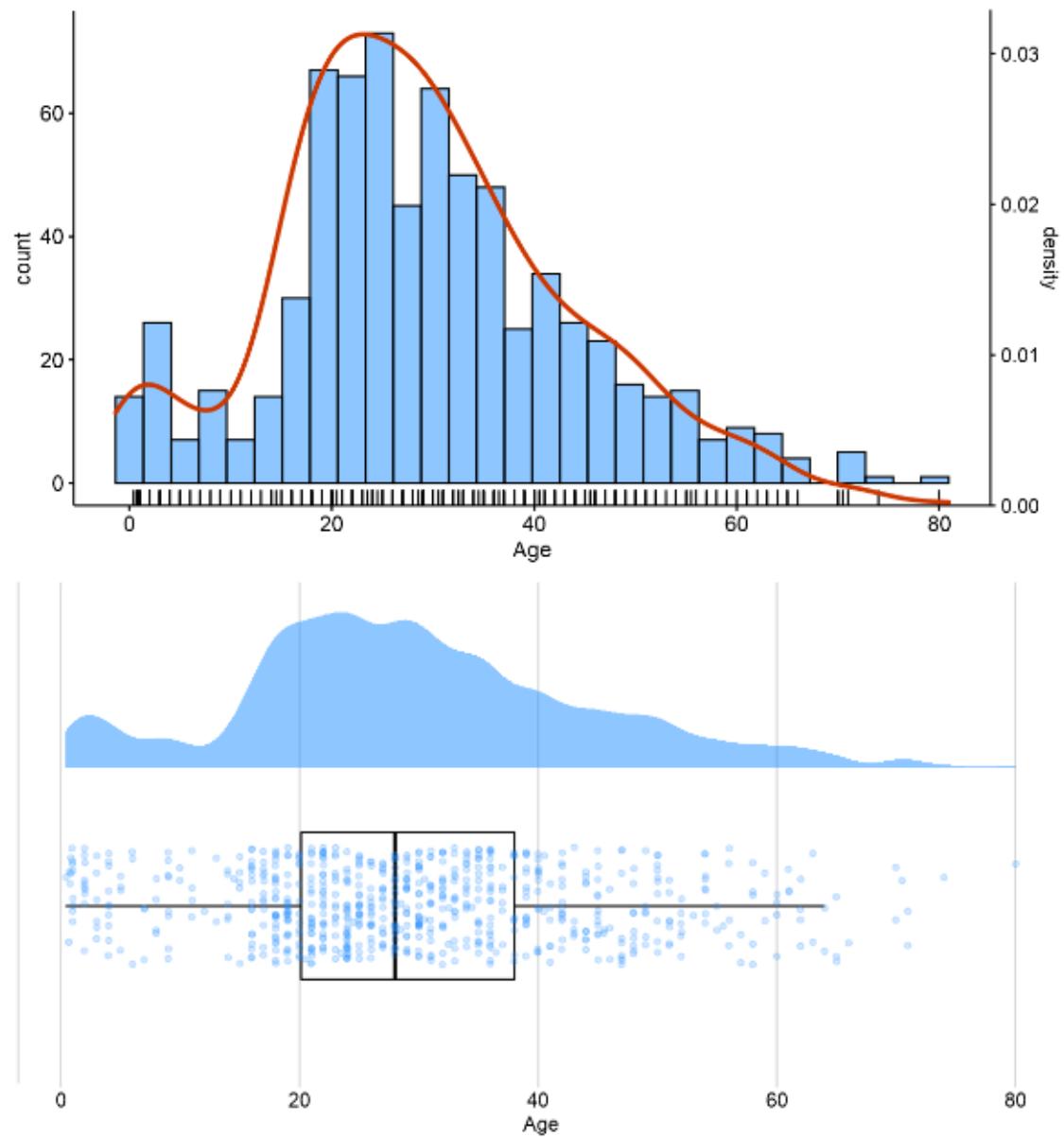


Figure 14.9 – Univariate analysis of the Age variable

17. If you select the yeo-johnson transformation from the **Transformations** dropdown, the plots will update accordingly:

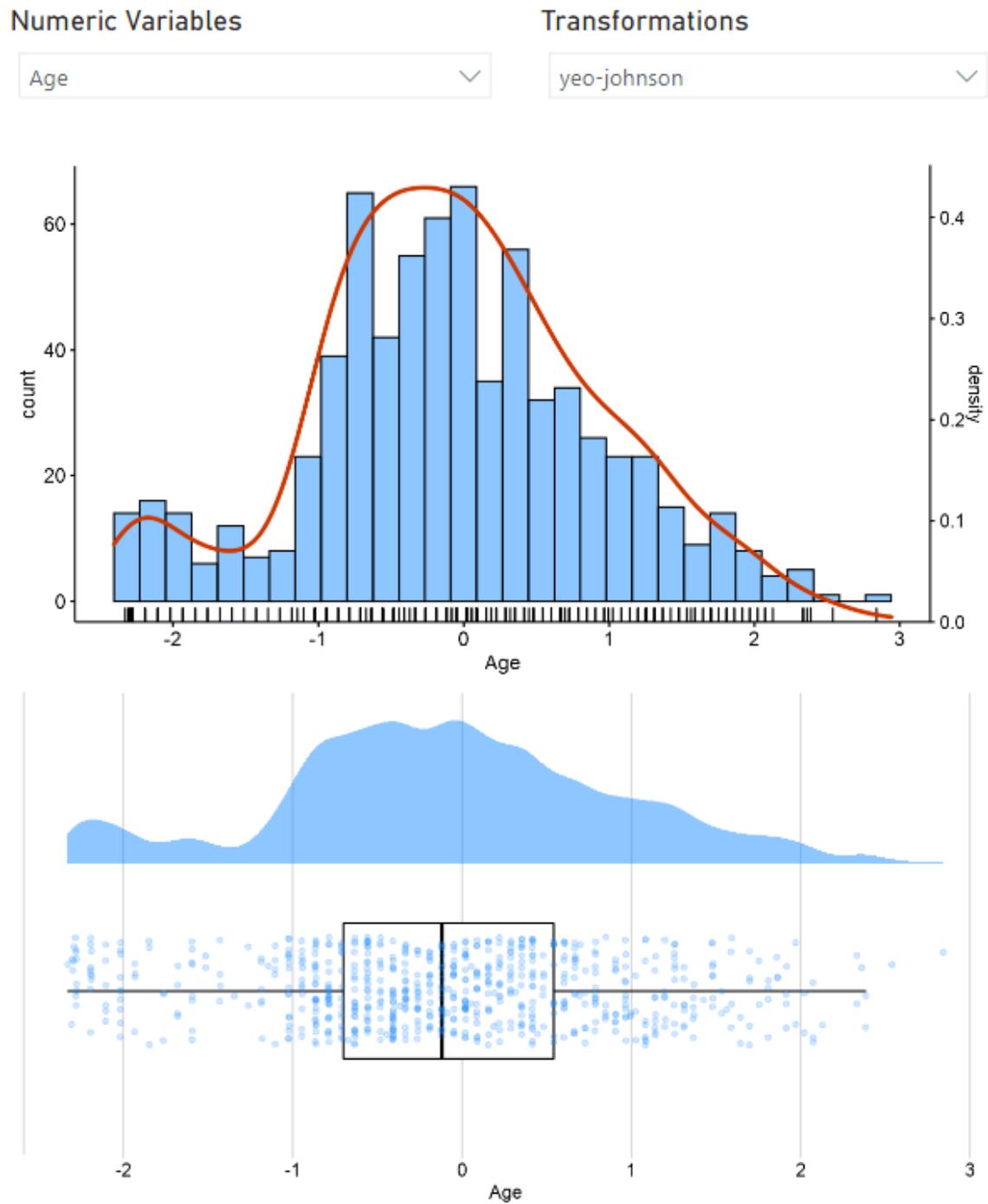


Figure 14.10 – Univariate analysis of the Yeo-Johnson transformed Age variable

18. Let's now create the slicer of categorical variables. Click on an empty spot in the report canvas first. Then click on the slicer visual icon.

Then expand the `categorical_df` table under the **Fields** panel and check the `categorical_col_name` measure.

19. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
20. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, switch off the **Slicer header** one, and add the **Title** Categorical Variables .
21. Now click on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane and enable it when you're prompted to do so. After that, move it under the slicer you've just created.
22. Keeping it selected, click on the field of the `categorical_df` table (`categorical_col_name`).
23. Now click on the R Visual's **Format** tab and switch off the **Title** option.
24. Copy the code of the file `06-plot-categorical-variables.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top-right of the R script editor (enable the R Visual every time it's requested). You'll get the univariate analysis of the `Cabin` variable:

Categorical Variables

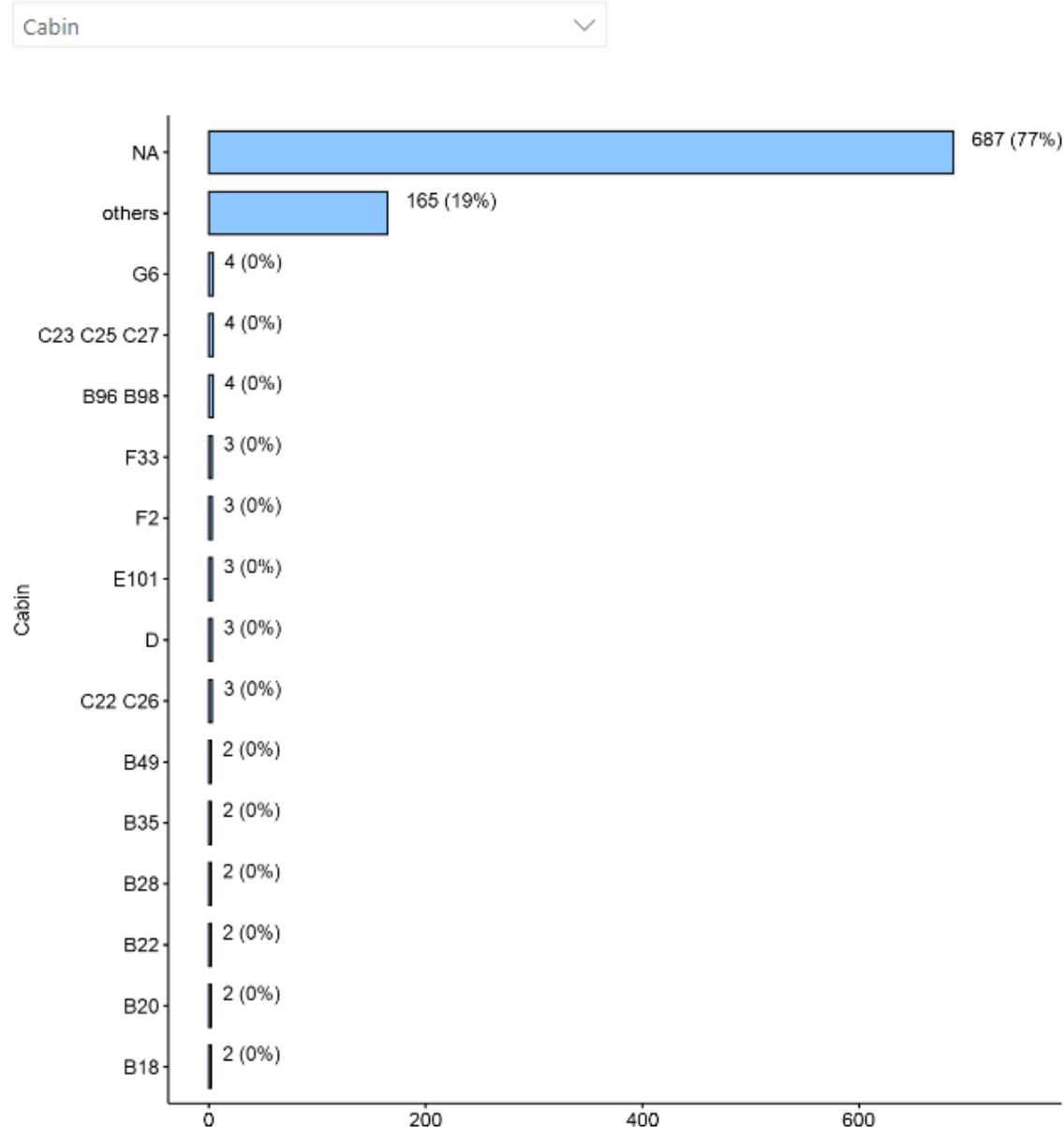


Figure 14.11 – Univariate analysis of the Cabin variable

25. Just rename the current page (right-click on the tab at the bottom of the page and click **Rename Page**) using the label **Univariate Analysis**.

Nice one! You just developed a great EDA report page dedicated to univariate dataset analysis. Playing with the slicers, the whole page will be like the following one:

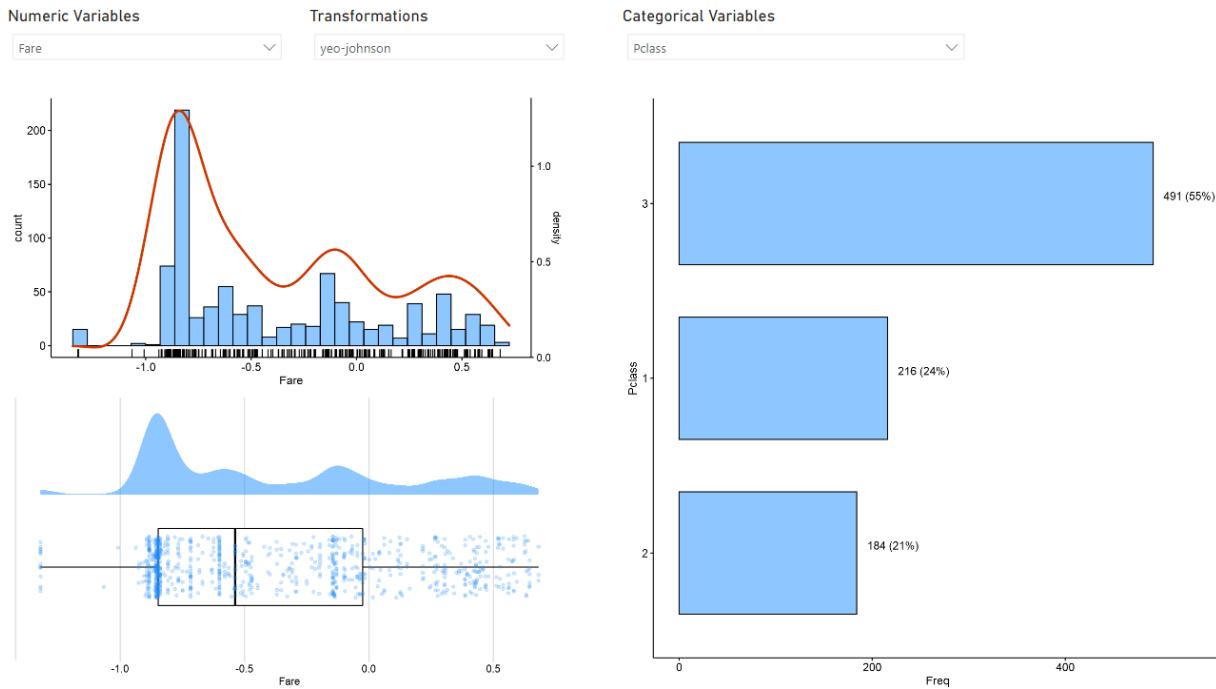


Figure 14.12 – The Univariate Analysis EDA report’s page

It is now time to develop a page of the report dedicated to multivariate analysis of the dataset.

Multivariate exploration

In general, the analysis of variables taken individually is never sufficient to give the analyst a complete overview of the dataset. Being able to use visuals that allow the interaction of multiple variables depending on their data type is definitely a step towards more accurate analysis of the dataset.

The most common visual used for analyzing the interaction of two numerical variables is the **scatterplot**. In our case, we will use an enriched version of this plot, adding the **marginal distributions** of the variables at the edges. Moreover, it’s possible to color the points according to another categorical variable. In this way, the analyst can study the interaction of three variables. In *Figure 14.13*, you can find an example of this plot for the variables `Age` and `Fare` grouped by the `Sex` one:

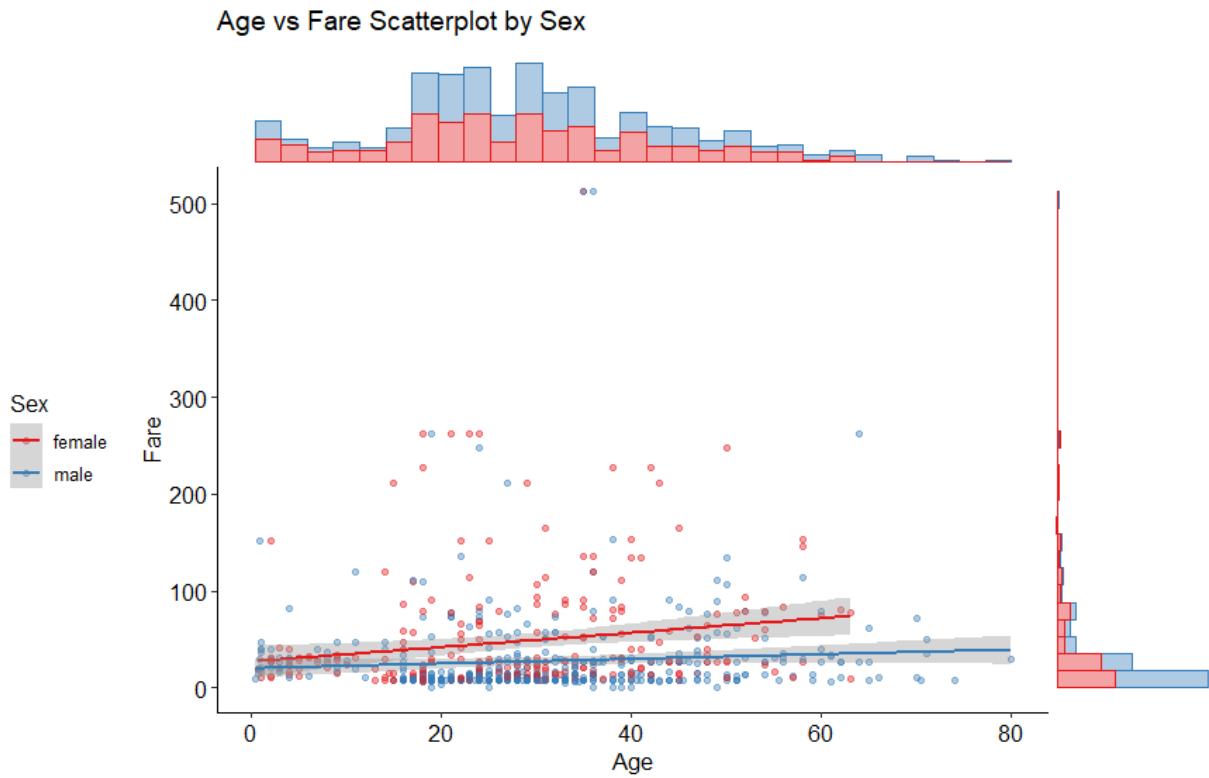


Figure 14.13 – Scatterplot with marginal histograms for Age and Fare by Sex

As you can see, there are as many regression lines as there are categories of the grouping variable. In the event of a missing grouping variable, the regression line is unique and in that case, the plot also shows the values of Pearson's R coefficient, Spearman's **rho**, and Kendall's **tau**, with the p-value derived from the respective tests. In short, in this case, the **p-value** represents the probability that the correlation between x and y in the sample data occurred by chance (take a look at the references to learn more about p-values):

Age vs YeoJohnson(Fare) Scatterplot

$r = 0.130, p = 0.001$ | $\rho = 0.135, p = < 0.001$ | $\tau = 0.093, p = < 0.001$

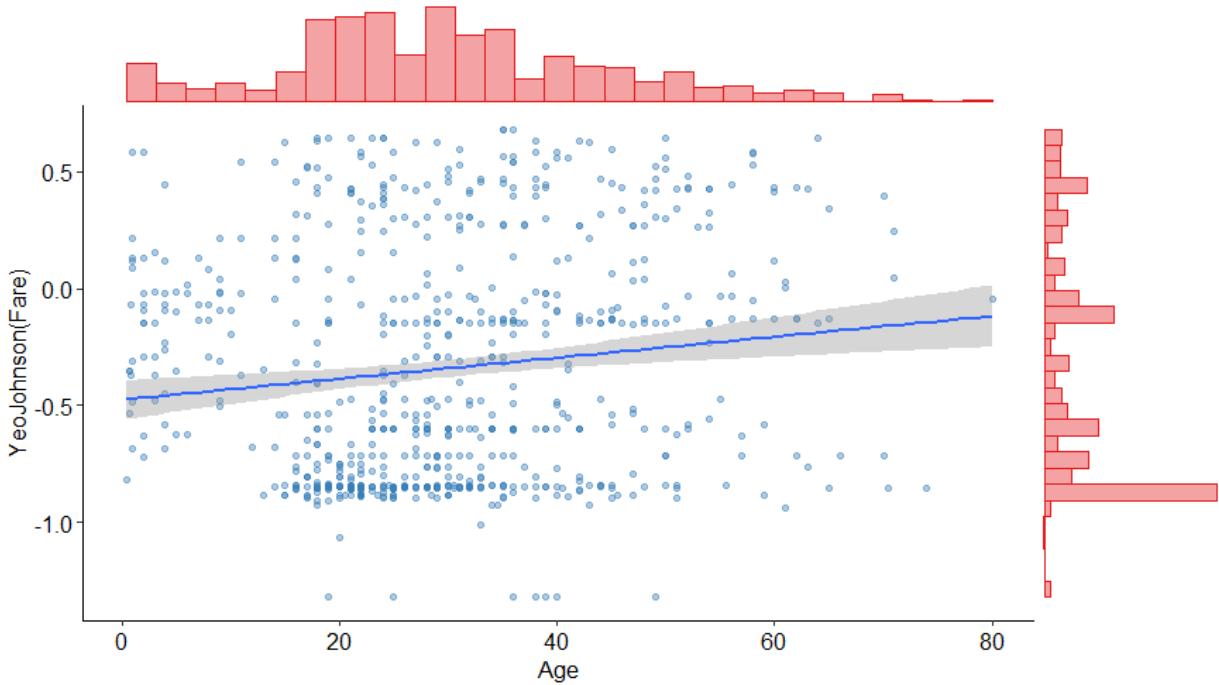


Figure 14.14 – Scatterplot for Age and Fare (transformed) without grouping

If the variables to study the interaction of are categorical, the **mosaic plot** comes to your aid. In *Figure 14.15*, you can find the mosaic plot of the SibSp and Pclass variables grouped by the Sex one:

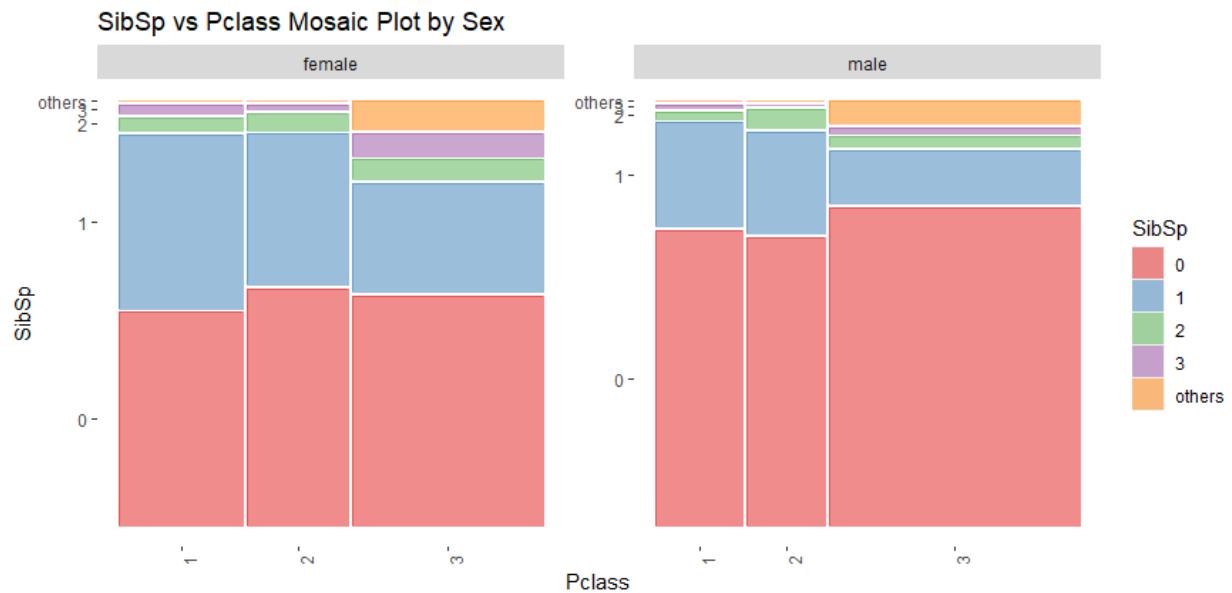


Figure 14.15 – Mosaic plot for Pclass and SibSp grouped by Sex

Finally, it is possible to study the interaction of a numerical variable with a categorical one by visualizing **raincloud plots for each category**. In addition, it is always possible to split the plot by a third categorical variable, as in the previous cases. In *Figure 14.16*, you can see an example of this plot for the variables `Fare` (transformed by Yeo-Johnson for a better visualization) and `Pclass` by the `Sex` one:

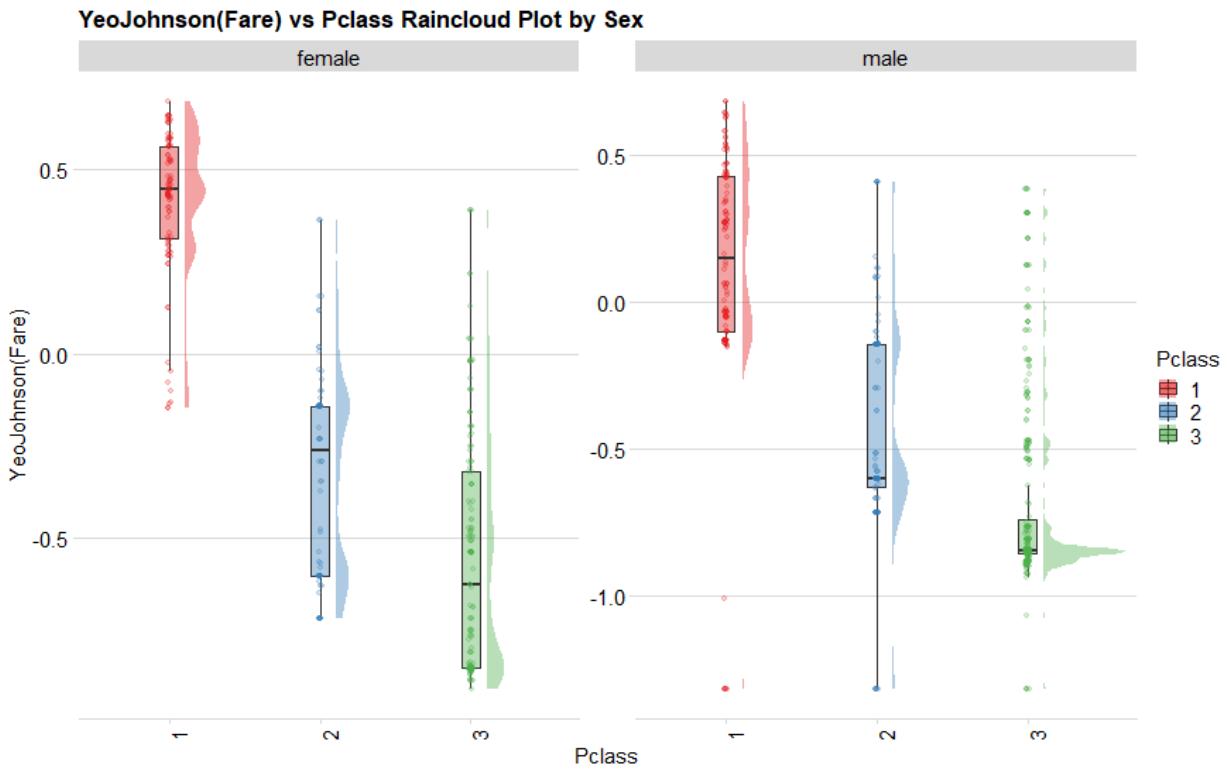


Figure 14.16 – Raincloud plots for the variables Fare and Pclass grouped by Sex

Now you can start developing the new page of your report dedicated to the univariate analysis of the dataset:

1. If it does not already exist, create a new Power BI report page clicking on the plus icon near the tabs at the bottom of the canvas. A new tab named **Page 1** will be created.
2. Click on **Get Data**, search for `script`, select **R script**, and click on **Connect**.
3. Enter the script you can find in the file `07-create-multivariate-objects.R` into the `Chapter14\R` folder. Make sure that the `folder` variable points to the correct absolute path of the `Chapter14` folder.
4. In the navigator dialog, select the `multivariate_df` table. This time, the code to generate the plots will be written directly in the R Visual, without the need to serialize them first and recall them at the time of visualization. This strategy is opposite to the one used for the page of

the univariate analysis because, in this case, the number of combinations between the variables in the game is much higher, and to generate the same number of plots in a list and to serialize everything is not very efficient. The `multivariate_df` table is a dataframe that is the result of a cross join between all the variables involved in the multivariate analysis (two numerical variables, a variable associated with the type of transformation for each of them, two categorical variables, and another categorical one for the grouping: seven variables in total):

$A_C^B x$	$A_C^B x_transf_type$	$A_C^B y$	$A_C^B y_transf_type$	$A_C^B cat1$	$A_C^B cat2$	$A_C^B grp$
Survived	standard	Survived	standard	Survived	Survived	Survived
Pclass	standard	Survived	standard	Survived	Survived	Survived
Age	standard	Survived	standard	Survived	Survived	Survived
SibSp	standard	Survived	standard	Survived	Survived	Survived
Parch	standard	Survived	standard	Survived	Survived	Survived
Fare	standard	Survived	standard	Survived	Survived	Survived

Figure 14.17 – `multivariate_df` table's content

Then click on **Load**.

5. Click on the slicer visual icon. Then expand the `multivariate_df` table under the **Fields** panel and check the `x` measure.
6. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
7. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the `x numeric` title under the **Slicer header** section, switch on the **Title** section, and add the `Analysis Variables` title.
8. Click on an empty spot in the report canvas first and then on the slicer visual icon. Expand the `multivariate_df` table under the **Fields** panel, and check the `x_transf_type` measure.
9. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
10. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the `x transformation` title under the **Slicer header** section, and keep the **Title** section switched off.
11. Repeat steps 5 to 10 in order to create the `y numeric` and `y transformation` slicers, making sure to select the `y` and

`y_transf_type` measures respectively. Keep the **Title** section switched off for them also.

12. Click on an empty spot in the report canvas first and then on the slicer visual icon. Expand the `multivariate_df` table under the **Fields** panel, and check the `cat1` measure.
13. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
14. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the `cat1` categorical title under the **Slicer header** section and keep the **Title** section switched off.
15. Repeat steps 12 to 14 in order to create the **cat2 categorical** slicer. Make sure to select the `cat2` measure for it.
16. Click on an empty spot in the report canvas first and then on the slicer visual icon. Expand the `multivariate_df` table under the **Fields** panel and check the `grp` measure.
17. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
18. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the `grp` categorical title under the **Slicer header** section, and keep the **Title** section switched off.
19. Try to align all these sliders on a single row on top of your page like the following:

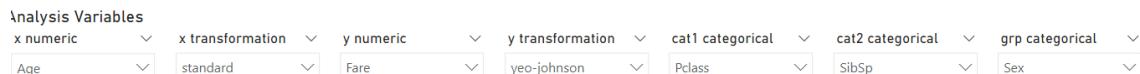


Figure 14.18 – Multivariate page's slicers on top of the page

20. Now click on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane and enable it when you're prompted to do so. After that, move it under the row of slicers you've just created and resize it in order to fill the page.
21. Keeping it selected, click on all the fields of the `multivariate_df` table.

22. Now click on the R Visual's **Format** tab and switch off the **Title** option.
23. Copy the code of the file `08-plot-multivariate-plots.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top right of the R script editor (enable the R Visual every time it's requested). You'll get all the multivariate analysis plots in it.
24. Just rename the current page (right-click on the tab at the bottom of the page and click **Rename Page**) using the label **Multivariate Analysis**.

Amazing! This EDA report page dedicated to multivariate analysis is very revealing for an analyst. Playing with the slicers, the whole page will be like the following one:

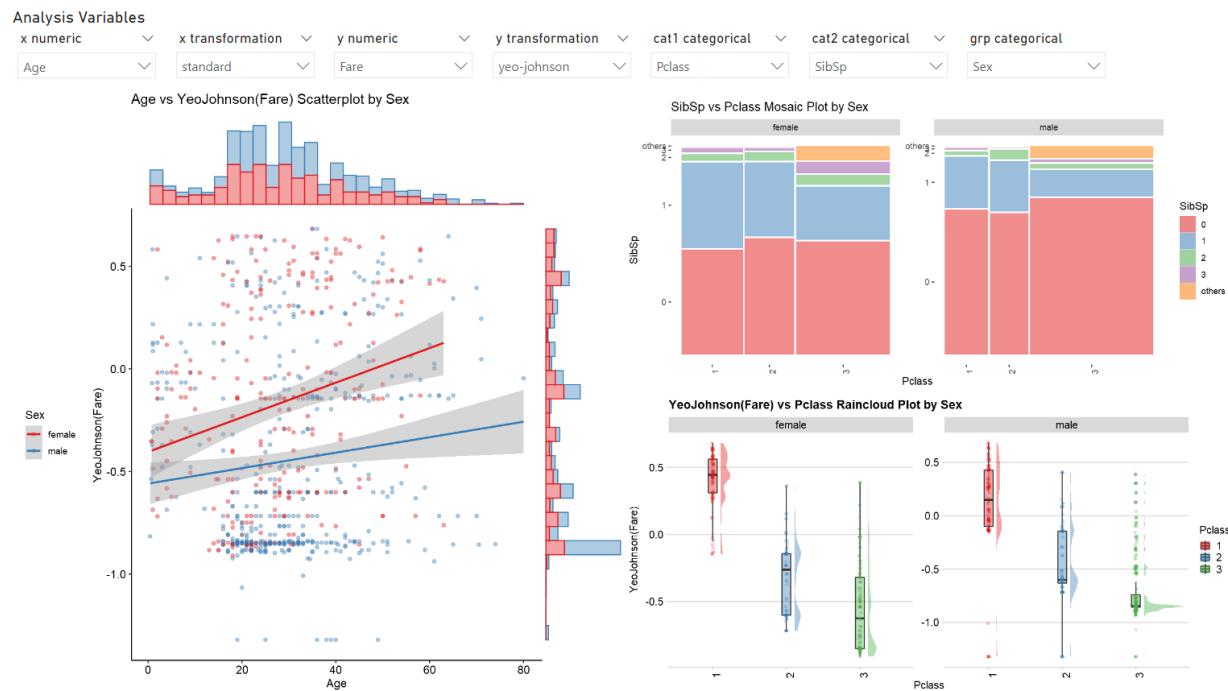


Figure 14.19 – Multivariate analysis page of the EDA report

Keep in mind that the numeric `y` variable is used for both the scatterplot and the raincloud plot as the `y` axis. The categorical `cat1` variable is used as the `x` axis for both the raincloud plot and the mosaic plot. The categorical `grp` variable is used to group all three plots by their labels.

You've just had a demonstration of how powerful and flexible R is for professional graphics development. Let's see how to develop a page for variable associations.

Variable associations

As you learned from *Chapter 11, Adding Statistics Insights: Associations*, knowing the correlation coefficient that binds two variables gives you the opportunity to assess the predictive power of one variable toward the other. From the same chapter, you also learned how to calculate the correlation coefficient depending on combinations of the different types of variables involved.

In this section, we will put what you have learned to use in developing a page for the EDA report containing a **heatmap plot** colored according to the intensity of the correlation coefficient. Here is an example:

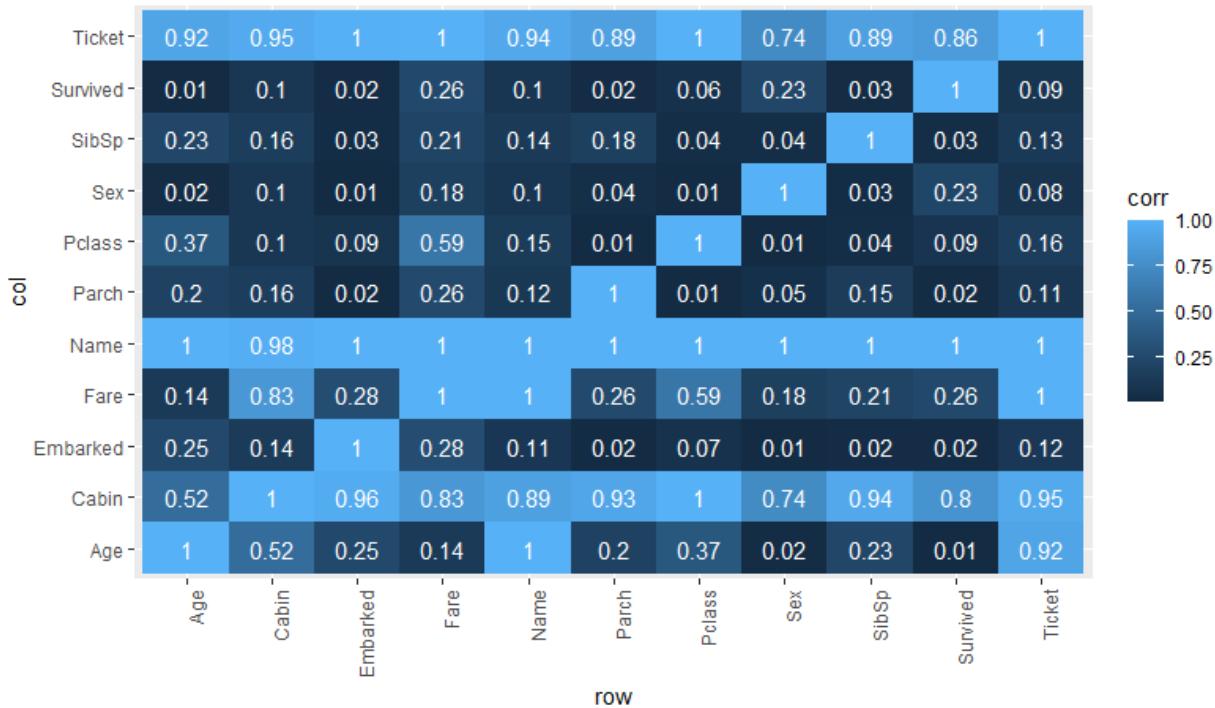


Figure 14.20 – Correlation heatmap of the dataset

The correlation coefficient will be calculated according to the selection that the analyst can make thanks to two slicers: one that allows you to choose

which method to use for the correlation between numerical variables, and another that allows you to choose the method to use for the correlation between categorical variables.

So, let's start to develop the **Associations** page:

1. If it does not already exist, create a new Power BI report page by clicking on the plus icon near the tabs at the bottom of the canvas. A new tab named **Page 1** will be created.
2. Click on **Get Data**, search for `script`, select **R script**, and click on **Connect**.
3. Enter the script you can find in the file `09-create-association-objects.R` into the `Chapter14\R` folder. Make sure that the `folder` variable points to the correct absolute path of the `Chapter14` folder.
4. In the **Navigator** dialog, select the `corr_tbl` table. This table is given by the set of several transformations. First, for each combination of correlation methods (for numerical and categorical variables), a column containing the respective correlation coefficient was calculated. This was sped up by using the parallel computation for R introduced in *Chapter 9, Calling External APIs To Enrich Your Data*. After that, a pivot operation was performed, bringing all these columns into the single column `corr` and creating the new column `corr_type`, which contains the string identifying the combination of methods used. Finally, the `corr_type` column was split into two separate columns, one for use in the correlation method slicer for numerical variables, the other for use in the correlation method slicer for categorical variables. Here is a sample of the result table:

corr_tbl

row	col	numeric_corr_type	categorical_corr_type	corr
Age	Age	pearson	theil	1
Age	Age	pearson	cramer	1
Age	Age	spearman	theil	1
Age	Age	spearman	cramer	1
Age	Age	kendall	theil	1
Age	Age	kendall	cramer	1
Age	Cabin	pearson	theil	0.524193719
Age	Cabin	pearson	cramer	0.524193719
Age	Cabin	spearman	theil	0.524193719

Figure 14.21 – corr_tbl table's content

Then click on **Load**.

5. Click on the slicer visual icon. Then expand the corr_tb1 table under the **Fields** panel and check the numeric_corr_type measure.
6. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
7. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the Numeric correlation type title under the **Slicer header** section, switch on the **Title** section, and add the Analysis Variables title.
8. Click on an empty spot in the report canvas first and then on the slicer visual icon. Expand the corr_tb1 table under the **Fields** panel and check the categorical_corr_type measure.
9. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
10. Resize the bottom edge of the slicer, click on its **Format** options, switch on the **Single select** one under **Selection controls**, enter the Categorical correlation type title under the **Slicer header** section and keep the **Title** section switched off.
11. Try to align all these sliders on a single row at the top of your page like the following:

The screenshot shows the 'Analysis Variables' section of a report canvas. At the top, there are two dropdown menus. The left menu is labeled 'Numeric correlation type' and has 'pearson' selected. The right menu is labeled 'Categorical correlation type' and has 'theil' selected. Both menus have a downward arrow icon to their right.

Figure 14.22 – Association page's slicers at the top of the page

12. Now click on an empty spot in the report canvas first. Then click on the **R Visual** icon in the **Visualizations** pane and enable it when you're prompted to do so. After that, move it under the row of slicers you've just created and resize it in order to fill the page.
13. Keeping it selected, click on the `col`, `corr`, and `row` fields of the `corr_tbl` table.
14. Copy the code of the file `10-plot-association-plots.R` into the `Chapter14\R` folder and paste it into the R Visual script editor. Then click on the **Run script** arrow icon at the top right of the R script editor (enable the R Visual every time it's requested). You'll get all the multivariate analysis plots in it.
15. Now click on the R Visual's **Format** tab and under the **Title** section, enter the string `Categorical Heatmap` as the title.
16. Just rename the current page (right-click on the tab at the bottom of the page and click **Rename Page**) using the label `Association Analysis`.

It's impressive how easy it was to create a correlation heatmap with R and Ggplot !

Playing with the slicers, the whole page will be like the following one:

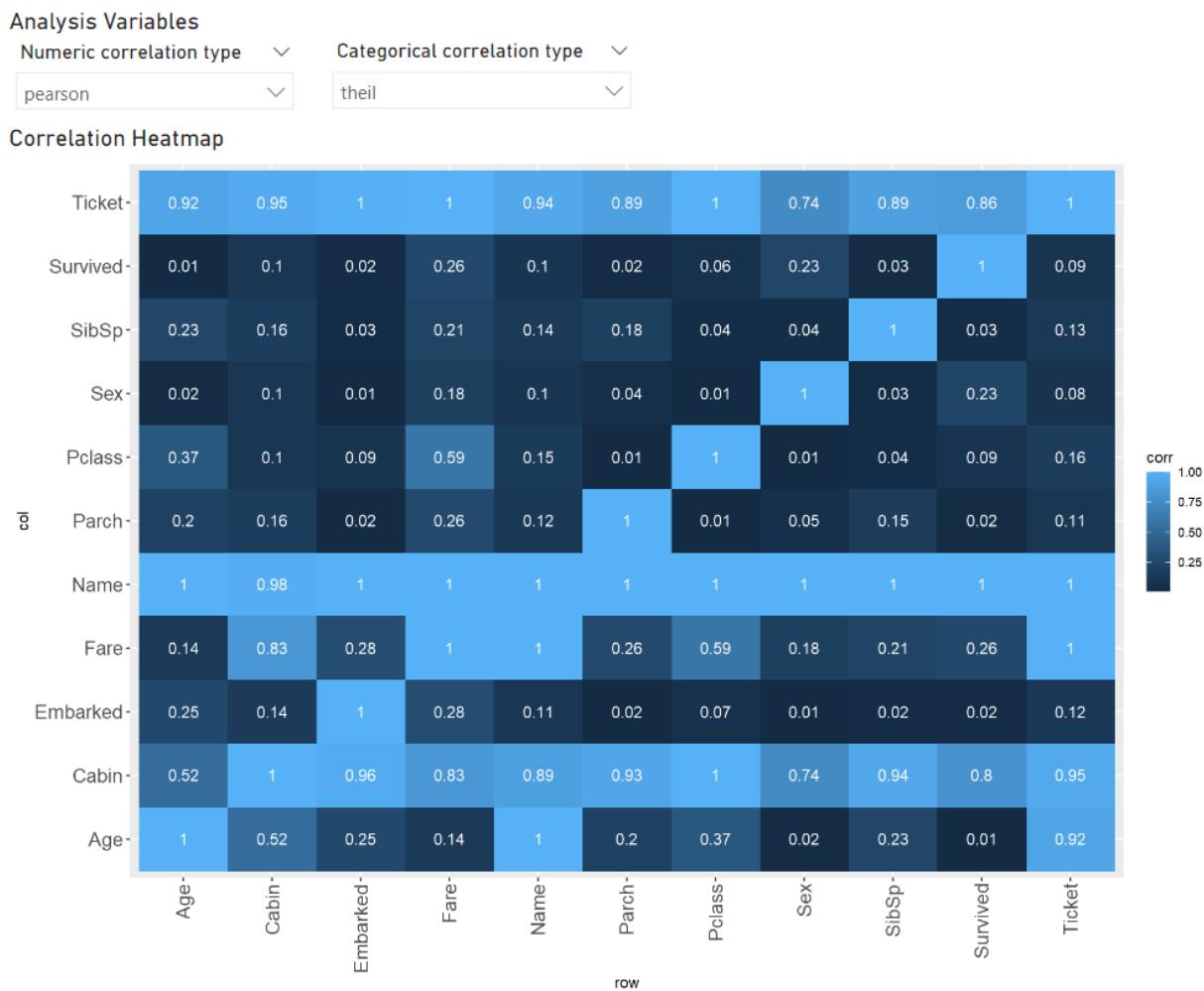


Figure 14.23 – Association analysis page of the EDA report

Wow! You have completed your first EDA report. You have to admit that the result is amazing as it is!

Summary

In this chapter, you learned what Exploratory Data Analysis (EDA) is for and what goals it helps achieve.

You also learned what tools are most commonly used to do automated EDA using Python and R.

Finally, you developed a complete and dynamic EDA report for analyzing a dataset using R and its most popular packages for creating graphics.

In the next chapter, you'll see how to develop advanced visualizations.

References

For additional reading, check out the following books and articles:

1. *Powerful EDA (Exploratory Data Analysis) in just two lines of code using Sweetviz* (<https://towardsdatascience.com/powerful-eda-exploratory-data-analysis-in-just-two-lines-of-code-using-sweetviz-6c943d32f34>)
2. *Lux: A Python API for Intelligent Visual Data Discovery* (https://www.youtube.com/watch?v=YANlids_Nkk)
3. *pandas Profiling of the Titanic Dataset* (https://pandas-profiling.github.io/pandas-profiling/examples/master/titanic/titanic_report.html)
4. *pandasGUI Demo* (<https://www.youtube.com/watch?v=NKXdolMxW2Y>)
5. *A Comprehensive Guide to the Grammar of Graphics for Effective Visualization of Multi-dimensional Data* (<https://towardsdatascience.com/a-comprehensive-guide-to-the-grammar-of-graphics-for-effective-visualization-of-multi-dimensional-1f92b4ed4149>)
6. *The Complete ggplot2 Tutorial* (<http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html>)
7. *Everything you need to know about interpreting correlations* (<https://towardsdatascience.com/eveything-you-need-to-know-about-interpreting-correlations-2c485841c0b8>)

15 Advanced Visualizations

As you have already seen in *Chapter 14, Exploratory Data Analysis*, thanks to the `ggplot2` package and its extensions, you can create very professional-looking charts. In this chapter, you'll see how you can create a very advanced and attractive custom chart. You will learn about the following topics:

- Choosing a circular barplot
- Implementing a circular barplot in R
- Implementing a circular barplot in Power BI

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** already installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

Choosing a circular barplot

Very often, it happens that we need to represent the measures associated with various categorical entities using a **bar chart** (or **barplot**). However, when the number of entities to be represented exceeds 15 or 20, the graph starts to become unreadable, even though you arrange it vertically:

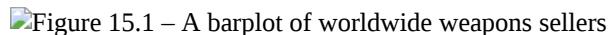
Figure 15.1 – A barplot of worldwide weapons sellers

Figure 15.1 – A barplot of worldwide weapons sellers

In this case, as you have already seen in *Chapter 14, Exploratory Data Analysis*, it is often a good idea to represent a maximum number of entities, after which the subsequent individual entities are grouped into a single category (in our case, the *Others* category). In this way, the readability of the graph is preserved, but part of the information you want to represent is lost.

If it is strictly necessary to display all entities with their measure, we often resort to a more eye-catching organization of the space occupied by the barplot, wrapping it in a circular shape, thus obtaining a **circular barplot**:

Figure 15.2 – Circular barplot of worldwide weapons sellers

Figure 15.2 – Circular barplot of worldwide weapons sellers

The graph becomes more interesting when you need not only to represent all entities but also to group them appropriately. Consider having a list of 24 speakers at an event, each of whom has received feedback (1 to 5) from viewers on some of their characteristics shown during the speech (*Expectation*, *Interesting*, *Useful*, and *Oral Presentation*). So, in the end, you have $24 \times 4 = 96$ entities to show. The grouped circular barplot describing this situation is very illustrative:

Figure 15.3 – A circular barplot with groups of speaker feedback

Figure 15.3 – A circular barplot with groups of speaker feedback

The circular barplot in *Figure 15.3* is the one we're going to implement using R and the `ggplot2` package in the next section.

Implementing a circular barplot in R

The R code you will find in this section is inspired by the code shared with the entire R community by the **R Graph Gallery** website (<https://www.r-graph-gallery.com/index.html>). In addition to a few very small additions, we refactored and generalized the code into the `circular_grouped_barplot()` function using the `tidy` evaluation framework (check the references for further details) so that it can be used with any dataset.

If you remember correctly, in R functions you saw in previous chapters, you passed column names to functions as strings. Thanks to `tidy` evaluation, you can pass them to functions using *tidyverse grammar*, that is, passing them directly through a pipeline. Take the following example:

```
circular_grouped_barplot(data = speakers_tbl,
                           grp_col_name = 'Characteristics',
                           label_col_name = 'SpeakerName',
                           value_col_name = 'Value')
```

Instead of having that call for the previous function, you will have the following one:

```
speakers_tbl %>%
  circular_grouped_barplot(grp_col_name = Characteristics,
                            label_col_name = SpeakerName,
                            value_col_name = Value)
```

Note how in the last script, the quotes identifying the strings have disappeared.

Let's take a step-by-step look at what this function does. The code you will find here is excerpted from the `01-circular-grouped-barplot.R` file you can find in the `Chapter15` folder. Let's proceed as follows:

1. First, make sure the `scales` package is already installed in your CRAN R dedicated to R visuals (in our case, version 3.4.4). Just check whether its name is in the list you can find in the bottom-right **Packages** tab in RStudio. If not, install it as always, using the `install.packages('scales')` command.
2. The dataset you will use is taken from the `Scores.csv` file you can find in the `Chapter15` folder. As mentioned in the previous section, it contains the average feedback obtained by 24 speakers from attendees at a conference on some of their characteristics shown during the speech. The tibble looks like the following:

Figure 15.4 – The speakers tibble
Figure 15.4 – The speakers tibble

3. Immediately after the call to the `circular_grouped_barplot()` function, the internal `rescale100()` function is defined, which is used to rescale the values of the entire dataset (in our case, the votes) on a scale of 0 to 100.
4. The execution of the arguments passed to the function via the `enquo()` function is then delayed (for more details, see the references):

```
grp_var <- enquo(grp_col_name)
label_var <- enquo(label_col_name)
value_var <- enquo(value_col_name)
```

5. The categorical variables of the input dataset are transformed into factors, and the values are scaled thanks to the function defined previously, all using `tidy` evaluation :

```
data <- data %>%
  mutate(
    !!quo_name(grp_var) := as.factor (!!grp_var),
    !!quo_name(label_var) := as.factor (!!label_var),
    !!quo_name(value_var) := rescale100 (!!value_var) )
```

6. In order to separate each group of bars, a few empty bars are added at the end of each group. First, an empty bars DataFrame is defined:

```

empty_bars <- 3
# Create the empty dataframe to add to the source dataframe
data_to_add <- data.frame(matrix(NA, empty_bars * nlevels(data[[quo_name(grp_var)]])), ncol(d
colnames(data_to_add) <- colnames(data)
data_to_add[[quo_name(grp_var)]] <- rep(levels(data[[quo_name(grp_var)]])), each = empty_bars

```

It has the following shape:

Figure 15.5 – The empty bars DataFrame
Figure 15.5 – The empty bars DataFrame

7. Then, the empty DataFrame is added to the source DataFrame (the `data` one). Once the resulting DataFrame is reordered for the grouping variable and values, the rows of the empty bars DataFrame are automatically distributed at the end of each group:

```

data <- rbind(data, data_to_add)
# Reorder data by groups and values
data <- data %>% arrange (!!grp_var, !!value_var)

```

8. A bar identifier is added:

```
data$id <- seq(1, nrow(data))
```

It is used to calculate the angle at which each label has to be displayed:

```

# Get the total number of bars
number_of_bars <- nrow(data)
# Subtract 0.5 from id because the label must have the angle of the center of the bars,
# Not extreme right(1) or extreme left (0)
angles_of_bars <- 90 - 360 * (data$id - 0.5) / number_of_bars

```

9. We define the DataFrame of the labels, starting from the `data` one, considering the correct alignment of the labels with the bars and the correct angle they must have to be readable:

```

label_data <- data
label_data$hjust <- ifelse( angles_of_bars < -90, 1, 0)
label_data$angle <- ifelse( angles_of_bars < -90, angles_of_bars + 180, angles_of_bars)

```

10. A DataFrame for base lines of groups is defined. It contains the start and end IDs (bars) of each group, the mean ID of each group (used as a base point for group text labels), and the angle at which each text label has to be rotated:

```

base_data <- data %>%
  group_by (!!grp_var) %>%
  summarize(start = min(id),
            end = max(id) - empty_bars) %>%
  rowwise() %>%
  mutate(title = floor(mean(c(start, end)))) %>%
  inner_join( label_data %>% select(id, angle),
              by = c('title' = 'id')) %>%
  mutate( angle = ifelse( (angle > 0 & angle <= 90) |
                           (angle > 180 & angle <= 270),
                           angle-90, angle+90 ) )

```

11. An `if` clause determines whether to initially define a barplot with only one group or with multiple groups, depending on the content of the source dataset. In our case, since there are four groups, the code is the following:

```

p <- data %>%
  ggplot(aes_string(x = 'id', y = quo_name(value_var), fill = quo_name(grp_var))) +
  # Add a barplot
  geom_bar(stat = 'identity', alpha=0.5) +
  # Add 100/75/50/25 indicators
  geom_segment(data = grid_data,

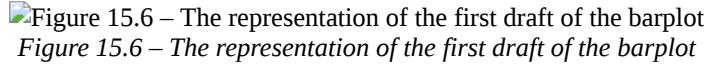
```

```

aes(x = end, y = 100, xend = start, yend = 100),
colour = 'grey', alpha=1, size=0.3 ,
inherit.aes = FALSE ) +
geom_segment(data = grid_data,
aes(x = end, y = 80, xend = start, yend = 80),
colour = 'grey', alpha=1, size=0.3 ,
inherit.aes = FALSE ) +
geom_segment(data = grid_data,
aes(x = end, y = 60, xend = start, yend = 60),
colour = 'grey', alpha=1, size=0.3 ,
inherit.aes = FALSE ) +
geom_segment(data = grid_data,
aes(x = end, y = 40, xend = start, yend = 40),
colour = 'grey', alpha=1, size=0.3 ,
inherit.aes = FALSE ) +
geom_segment(data = grid_data,
aes(x = end, y = 20, xend = start, yend = 20),
colour = 'grey', alpha=1, size=0.3 ,
inherit.aes = FALSE ) +
# Add text showing the value of each 100/75/50/25 lines
annotate('text', x = rep(max(data$id), 5),
y = c(20, 40, 60, 80, 100),
label = c('20', '40', '60', '80', '100') ,
color='grey', size=3,
angle=0, fontface='bold', hjust=1)

```

The barplot representation is as follows:


Figure 15.6 – The representation of the first draft of the barplot

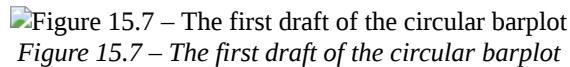
12. After defining a text justification vector, the previous plot is cleaned of unnecessary graphical frills, and it is magically wrapped into a circular shape simply by using the `coord_polar()` function:

```

p <- p +
# The space between the x-axis and the lower edge of the figure will implicitly define the
ylim(-120,120) +
theme_minimal() +
theme(
  legend.position = 'none',
  axis.text = element_blank(),
  axis.title = element_blank(),
  panel.grid = element_blank(),
  plot.margin = unit(rep(-1,4), 'cm')
) +
# Wrap all in a circle!
coord_polar()

```

What you get is the following first draft:


Figure 15.7 – The first draft of the circular barplot

Unfortunately, the `coord_polar()` function does not rotate or curve the labels. So, you have to add them separately, rotating them manually when needed.

13. Finally, let's add bar labels appropriately rotated, base lines of groups, and group text labels appropriately rotated:

```

p <- p +
# Add labels
geom_text(data = label_data %>% mutate( y_pos = !!value_var + 10),
aes_string(x = 'id', y = 'y_pos',
label = quo_name(label_var),
hjust = 'hjust'),

```

```

color = 'black', fontface = 'bold',
alpha = 0.6, size = 3,
angle = label_data$angle,
inherit.aes = FALSE) +
# Add base lines of groups
geom_segment(data = base_data,
aes(x = start, y = -5, xend = end, yend = -5),
colour = 'black', alpha=0.8,
size=0.6 , inherit.aes = FALSE ) +
# Add groups text
geom_text(data = base_data %>% mutate(y = -14),
aes_string(x = 'title', y = 'y',
label=quo_name(grp_var),
angle = 'angle'),
hjust = text_horiz_justification,
colour = 'black', alpha=0.8, size=4,
fontface='bold', inherit.aes = FALSE)

```

The result can be seen in *Figure 15.3*.

This is just one demonstration of how effective it is to make complex graphs with `ggplot2` by taking advantage of the *grammar of graphics*. As you have seen, this framework allows you to approach the creation of a complex chart by adding modular logic pieces one at a time, testing the result intuitively.

Important Note

The code used in this section uses an old version of `tidy evaluation`, tied to older versions of `rlang`, `dplyr`, and `ggplot2`. This choice is forced by the fact that the Power BI service uses an older version of the R engine (3.4.4) with older versions of the previous packages.

Starting with versions 4.0 of `rlang`, 3.0.0 of `ggplot2`, and 1.0.0 of `dplyr`, the syntaxes used for `tidy evaluation` have been simplified and standardized. You can find an example of the same function that draws a circular barplot translated for the new versions of the previous packages in the `02-circular-grouped-barplot-new-tidy-eval.R` file.

Let's see now how to implement the circular barplot in Power BI.

Implementing a circular barplot in Power BI

As you have already seen in the previous chapters, Power BI is capable of rendering graphs developed with `ggplot2` using R visuals. Therefore, whatever the complexity of the graph created using `ggplot2`, you can be sure that Power BI handles it well.

To create a circular barplot in Power BI, proceed as follows:

1. Make sure that Power BI Desktop is referencing the version of CRAN R dedicated to R visuals in **Options**.
2. Click on **Get Data**, **Text/CSV**, and then **Connect**.
3. Select the `Scores.csv` file found in the `Chapter15` folder and click **Open**.
4. You will see a preview of the CSV file. Make sure to select `65001: Unicode (UTF-8)` as **File Origin**. This way, special characters in speaker names will be displayed correctly. Then, click on **Load**.
5. Click on the **R Visual** icon in the **Visualizations** panel, enable it, and then resize the visual as all the available canvas.
6. Keeping the R visual selected, expand the `Scores` table under the **Fields** panel and check all its fields.
7. Click on the R visual's **Format** tab and switch off **Title**.
8. Copy the code of the `01-circular-grouped-barplot.R` file into the `Chapter15` folder and paste it into the R visual script editor. Then, click on the **Run script** arrow icon in the top right of the R script editor (enable the R visual every time it's requested). You'll get the circular barplot into it.
9. Click on an empty spot in the report canvas first and then click on the slicer visual icon. Then, expand the `Scores` table under the **Fields** panel and check the characteristic measure.

10. Click on the downward-pointing arrow at the top right of the slicer to select the **Dropdown** slicer type.
11. Resize the bottom and right edges of the slicer, move it to the center of the circular barplot, click on its **Format** options, and switch on **Show Select All option** under **Selection controls**.

You can now filter multiple characteristics (using the *Ctrl* key) and the circular barplot will update accordingly:

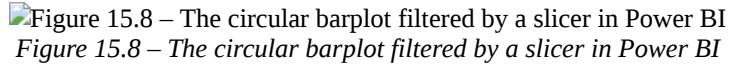


Figure 15.8 – The circular barplot filtered by a slicer in Power BI
Figure 15.8 – The circular barplot filtered by a slicer in Power BI

Very impressive, huh? By practicing a little bit with `ggplot2` and R, you can get as many impressive charts as you want to enrich your analysis.

Summary

In this chapter, you learned how to implement a circular barplot using R and `ggplot2`. You also had your first experience with `tidy` evaluation in refactoring the code that implements the circular barplot into a function.

After that, you implemented a circular barplot in Power BI.

In the next chapter, you'll learn how to develop interactive custom visuals in R and Power BI.

References

For additional reading, check out the following books and articles:

1. *Who Sells More Weapons?* (<https://www.data-to-viz.com/story/OneNumOneCat.html>)
2. *Circular barplot with groups* (<https://www.r-graph-gallery.com/297-circular-barplot-with-groups.html>)
3. *Down the rabbit hole with tidy eval — Part 1 (the old tidy eval way)* (<https://colinfay.me/tidyeval-1/>)
4. *Tidy evaluation (the new tidy eval way)* (<https://tidyeval.tidyverse.org/>)

16 Interactive R Custom Visuals

In *Chapter 15, Advanced Visualizations*, you saw that it is possible to make very complex graphs thanks to the flexibility introduced by **ggplot**. Sometimes, however, you have the feeling that you can't take full advantage of the information shown in the graph because of a lack interactivity, such as tooltips. In this chapter, you'll learn how to introduce interactivity into custom graphics created using R and by directly using *HTML widgets*. Here are the topics we will cover:

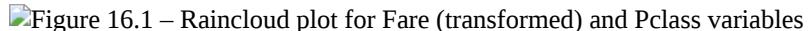
- Why interactive R custom visuals?
- Adding a dash of interactivity with Plotly
- Exploiting the interactivity provided by HTML widgets
- Packaging all into a Power BI custom visual
- Importing the custom visual package in Power BI

Technical requirements

This chapter requires you to have a working internet connection and **Power BI Desktop** installed on your machine. You must have properly configured the R and Python engines and IDEs as outlined in *Chapter 2, Configuring R with Power BI*, and *Chapter 3, Configuring Python with Power BI*.

Why interactive R custom visuals?

Let's start with a graph you've already implemented in R. Consider, for example, the raincloud plot of `Fare` vs `Pclass` variables introduced in *Chapter 14, Exploratory Data Analysis*:


Figure 16.1 – Raincloud plot for Fare (transformed) and Pclass variables

Focus for a moment only on the boxplots you see in *Figure 16.1*. Although the `Fare` variable is already transformed according to Yeo-Johnson to try to reduce skewness, there remain some extreme outliers for each of the passenger classes described by the categorical variable, `Pclass`. If, for example, you want to know the values of the transformed variable `Fare` corresponding to the whiskers (fences) of the boxplot on the left so that you can then determine the outliers located beyond those whiskers, it would be convenient that these values appear when you pass the mouse near that boxplot, as in *Figure 16.2*:


Figure 16.2 – Main labels shown in the Fare (transformed) boxplot for the first class

It would be even more interesting to know the actual value of a specific isolated outlier when you hover the mouse over the point representing it, as in *Figure 16.3*:


Figure 16.3 – Values of Fare (transformed) and Pclass for the highlighted outlier

There is no doubt that these interactivities would be welcomed by the analyst reading the charts if they were introduced.

So, let's see how to add these interactivities to an existing graphic developed using ggplot.

Adding a dash of interactivity with Plotly

There is an open source JavaScript library for data visualization, which is declarative and high-level and allows you to create dozens of types of interactive graphs, named **Plotly.js**. This library is the core of other Plotly client libraries, developed for Python, Scala, R, and ggplot. In particular, the library developed for R, named **Plotly.R** (<https://github.com/ropensci/plotly>), provides the `ggplotly()` function, which does all the magic for us: it detects all the basic attributes contained in an existing graph developed with ggplot and transforms them into an interactive web visualization. Let's see an example.

First, you need to install the Plotly.R library (<https://github.com/ropensci/plotly>) on your latest CRAN R engine via the `install.packages('plotly')` script.

Important Note

For simplicity, we'll make sure to run the custom visual on the latest version of the CRAN R engine, since all the necessary libraries were already installed in the previous chapters. If the goal is to publish a report with the custom visual on Power BI, you have to make sure that the custom visual is correctly rendered on the same R engine present on Power BI (in our case, CRAN R 3.4.4).

Then run the script you can find in the `01-interactive-boxplots.R` file in the `Chapter16` folder. The content of the script is extrapolated from the various scripts used in *Chapter 14, Exploratory Data Analysis*, so there's nothing new to you. The only portion of the script you haven't seen before is the following:

```
plotly::ggplotly(rc, tooltip = c('x', 'y'))
```

It is the part of code that transforms a static graph into a dynamic HTML-based one, and as you can see, it's one simple call to the `ggplotly()` function. The result is as follows:

Figure 16.4 – Result of applying the ggplotly() function to the raincloud plot
Figure 16.4 – Result of applying the ggplotly() function to the raincloud plot

As you can see, in RStudio the result is no longer shown in the **Plots** tab, but in the **Viewer** tab, dedicated to HTML output. You will also notice the presence of the **Modebar** in the top-right corner, which allows you to set some operations on the chart, such as zoom and hover options.

But the most striking thing is that you no longer see a series of raincloud plots, as you expected, but simple boxplots! If you look at the RStudio console, you will notice that there are three identical warning messages, one for each raincloud plot that should have been represented:

```
In geom2trace.default(dots[[1L]][[3L]], dots[[2L]][[1L]], dots[[3L]][[1L]]) :  
  geom_GeomSlabinterval() has yet to be implemented in plotly.  
  If you'd like to see this geom implemented,  
  Please open an issue with your example code at  
  https://github.com/ropensci/plotly/issues
```

You may have noticed that only the density plots have disappeared. This means that in the current version of Plotly (4.9.4.1), the objects created by the `ggdist` library are not yet managed. Moreover, `ggplotly()` renders fill and color aesthetics as distinct, as opposed to `ggplot` static graphs. Surely this functionality still needs to be improved and, most likely, these bugs will be handled in future versions. In any case, the graph is still usable and shows how easy it is to add interactivity to a ggplot graph.

At this point, you would think that any chart made in HTML and JavaScript could then be used in Power BI. Obviously, this is not the case.

Important Note

Every interactive graphic that can be used in Power BI must be an *HTML widget*.

In fact, Plotly.R exposes graphs via HTML widgets. Let's see what this is all about.

Exploiting the interactivity provided by HTML widgets

HTML widgets are R packages that allow you to build interactive web pages. These packages are generated by a framework used to create a binding between R and JavaScript libraries. This framework is made available by the `htmlwidgets` package developed by RStudio. HTML widgets are always hosted within an R package, including the source code and dependencies, in order to make sure that the widgets are fully reproducible even without being able to access the internet. For more details on how to develop an HTML widget from scratch, take a look at the references.

In addition to being able to embed HTML widgets in **RMarkdown** files (dynamic documents with R) or **Shiny** applications (interactive web apps built directly from R), the `htmlwidgets` package allows you to save them also in standalone web page files thanks to the `saveWidget()` function.

That said, there are hundreds of R packages that expose their functionalities in HTML widgets. You can explore the **htmlwidgets gallery** (<http://gallery.htmlwidgets.org/>) to search for interactive graphics that may be appropriate for you.

Okay, it is possible to visualize these HTML widgets, both those in the gallery and those made through Plotly, in Power BI. That sounds good. But how do you embed a dynamic graph made in HTML and JavaScript into Power BI? You must compile an R-powered visual via the *pbviz tools*.

Let's see how it's done.

Packaging it all into a Power BI Custom Visual

Power BI Visual Tools (pbviz) are the easiest way to build custom visuals in Power BI. They are written in **JavaScript** (using **Node.js**) and are used to compile the source code of **.pbviz packages**. A `.pbviz` package is a zipped version of the **Power BI Visual Project**, which in turn is a set of folders, scripts, and assets needed to create the custom visualization you want to implement. Generally, a standard Power BI Visual Project is created from a template thanks to the `pbviz` command-line tools. The template contents depend on the method by which you want to create the custom visual (**TypeScript**, **R Visual**, or **R HTML**).

Important Note

The `pbviz` tools do not support any technology that uses Python behind the scenes, such as the *ipywidget* widgets.

In the light of this, it is worth learning R and `ggplot` a little more in order to be able to develop interesting custom visuals using the R Visual and R HTML modes. In addition to this, as mentioned at the end of *Chapter 2, Configuring R With Power BI*, take the following note into account.

Important Note

The *Publish to web* option in Power BI does not allow you to publish reports that contain standard R Visuals. You can work around this limitation by creating R custom visuals, which are allowed to be published with this option.

Now let's see how to install the `pbviz` tools.

Installing the pbviz package

`pbviz` command-line tools provide everything you need to develop visuals and test them in reports and dashboards on Power BI service. For this very last reason, you need to install an SSL certificate too, so that your laptop can interact securely with the Power BI service.

Let's see how to do everything step by step:

1. Go to <https://nodejs.org/en/> and install the Node.js version recommended for all users, following the default options set by the installer.
2. Restart your laptop, as this is mandatory for the *Step 4* command to work.
3. Click on the Windows **Start** button and start entering the string “power”, then click on the **Windows PowerShell** app.
4. Enter the following command in the PowerShell console: `npm i -g powerbi-visuals-tools`. If you get some deprecation warnings, don’t worry and wait for the installation to complete.
5. Enter the following command in the PowerShell console: `pbviz --install-cert`. It returns a location in which the PFX file is created (it should be `C:\Users\<your-username>\AppData\Roaming\npm\node_modules\powerbi-visuals-tools\certs`) and a numeric passphrase. Take note of both as you’ll need them later.
6. Press **Win+R** and enter `mmc` in the **Open** textbox. It will open the **Microsoft Management Console**.
7. Click on **File**, then click on **Add/Remove Snap-in....**
8. Select **Certificates** and click on **Add**, then select **My user account** and click **Finish**.
9. Press **OK** on the **Add or Remove Snap-in** window.
10. In the main window, expand the **Certificates** node and the **Trusted Root Certification Authorities** one, select **Certificates** under this node. You’ll see all the certificates listed in the middle panel. On the **Action** panel on the right, click on **More Actions**, select **All Tasks...** and the click on **Import....**
11. Click **Next** on the **Certificate Import Wizard** welcome window. Click **Browse** on the next window and navigate to the location you noted in *Step 5*. Select **Personal Information Exchange (*.pfx, *.p12)** from the file type combobox near the **File name** textbox. Then the `PowerBICustomVisualTest_public.pfx` file will appear. Select it and click **Open**. Then click **Next**.
12. Enter the numeric password you noted in *Step 5*, keep the default **Import options** and click on **Next**.
13. Keep **Trusted Root Certification Authorities** as **Certificate store**, click **OK**. Click **Next** on the main window, then click **Finish**. You’ll get a **Security Warning** window asking if you want to install the certificate. Click **Yes** and a **The import was successful** dialog box will appear. Click **OK**.
14. In order to verify that everything went well, go back to the PowerShell console, enter the `pbviz` command and press **Enter**. You should see the following output:


Figure 16.5 – pbviz is installed correctly

Great! Now you have your pbviz tools properly configured and ready to compile a custom visual.

Let's put them to the test right now with an R HTML custom visual.

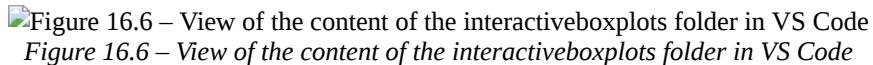
Developing your first R HTML custom visual

In order to create an R HTML custom visual, you must first generate a standard Power BI Visual Project of R HTML type starting from the template provided by pbviz tools. Then it is sufficient to modify the scripts provided by the project in order to create the visual you want to develop.

In this section, you will package the dynamic boxplots graph that you met in a previous section.

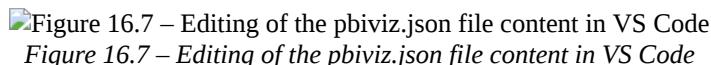
Here are the steps to obtain the `.pbviz` package:

1. Open the **Windows PowerShell** console from the **Start** menu if you have closed it since the last use. The folder in which the console will start by default is `C:\Users\<your-username>`.
2. Create a folder dedicated to custom visuals called `Power-BI-Custom-Visuals` using the `md Power-BI-Custom-Visuals` command.
3. Move into the folder you just created using the `cd Power-BI-Custom-Visuals` command.
4. Generate a standard R HTML Power BI Visual project from the template by using the `pbviz new interactiveboxplots -t rhtml` command.
5. Open the template folder you just created in VS Code. You will see something like this:



You will find a complete project ready to be compiled in the Chapter16\PBI-Visual-Project\interactiveboxplots folder in the GitHub repository. You can use it as a reference for the next steps.

6. Open the pbviz.json file of your just-created template to enter the basic information about the custom visual you are going to compile. Format it properly (right-click on the document and click on **Format Document**). In *Figure 16.7* and in the following figures you find on the left a part of the template code, and on the right, how it should be modified:



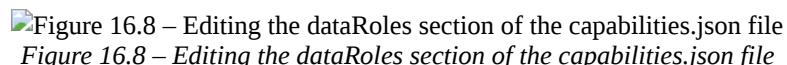
Here are the details of attributes to edit for the "visual" node:

- "name" : <name-of-your-custom-visual>
- "displayName" : <name-to-display-of-your-visual>
- "description" : <description-of-your-custom-visual>
- "supportUrl" : <url-to-be-contacted-for-support>

Here are the details of attributes to edit for the "author" node:

- "name" : <full-name-of-the-author>
- "email" : < email-of-the-author >

7. Now open the capabilities.json file. It is used to declare what data types the visualization accepts, what customizable attributes to put in the properties panel, and other information needed to create the visualization. It contains several root objects. The first one whose contents you need to edit is `dataRoles` (<https://bit.ly/pbiviz-dataroles>). In this section, you can define the data fields that your visual expects. The default template has just the unique `values` field as default, like the standard R Visual. In our case, the multivariate boxplot visual needs three fields:



8. Based on the items added in the `dataRoles`, you have to change the `capabilities.json` file's `dataViewMappings` root object content accordingly (<https://bit.ly/pbiviz-dataviewmappings>). They describe how data roles relate to each other and allow you to specify conditional requirements for data visualization. In our case, we need to declare the three fields created in the `dataRoles` as components of the script's input dataset:

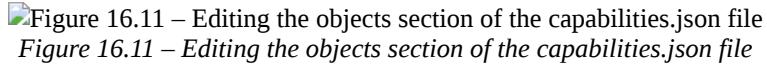


As you can see, the "script" subsection the template refers to the `rcv_script` object. We will see that it is defined in the next section.

9. The `capabilities.json` file's `objects` section describes the customizable properties that are associated with the visual and that appear in the **Format** pane (<https://bit.ly/pbiviz-objects>). In our case, we want to parameterize the type of transformation we can apply to the variable `y`. Therefore, we will make sure that the user can select the transformation type from the **Y Transformation Type** combobox present in the **Variables Settings** section just below the **General** section:

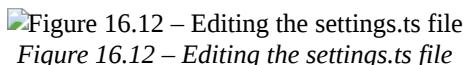


The changes to the script needed to achieve what you see in *Figure 16.10* is as follows:



The `suppressDefaultTitle` parameter allows you to suppress the title that generally appears at the top left of each R Visual. As you can see in *Figure 16.11*, the `rcv_script` object referenced in the `dataViewMappings` section is defined in this section. Contrary to the one you just added, the `rcv_script` object is not to be displayed in the **Format** pane, but is only used to describe the attributes of the `source` and `provider` objects that define the R script. In order to actually declare what parameters are to be displayed in the **Format** pane, you need to make a small change to the `settings.ts` file. Let's see how.

10. Open the `settings.ts` file found in the `src` folder of your Power BI Visual project. It contains the settings in TypeScript for the elements to be displayed in your visual. Instead of displaying the `rcv_script` object, we can display the `settings_variables_params` object that contains our parameter associated with the type of transformation to be applied to the `y` variable:



For further details about the classes used in this script, take a look at the references.

11. Open the `dependencies.json` file. It contains a reference to each library used in the R code that generates the visual. In addition to the ones already present (`ggplot2`, `plotly`, `htmlwidgets`, and `xml2`), you also need to add the following: `RColorBrewer`, `cowplot`, `dplyr`, `purrr`, `forcats`, and `recipes`. Just follow the syntax already used for the existing libraries, keeping in mind that you can put any string in `displayName`.
12. Finally, you can enter the R code that generates the visual in the `script.r` file. You can replace its entire content with that of the file of the same name that you find within the Power BI Visual project shared in the GitHub repository. At the beginning of the script, you will find some commented rows used to debug any issues in RStudio. Then there is a `source()` command that will load the utility functions from the provided `flatten_HTML.r` file in the `r_files` folder. They help to convert Plotly or widget objects to self-contained HTML. The next code is very similar to what you've already seen in the previous sections. There are integrated pieces of code to handle the presence of the fields passed to the visual as input data and the parameter that handles the type of transformation of the variable `y`. Here's an example:

```
y_transf_name <- 'standard'
if(exists("settings_variable_params_y_transf_name")){
  y_transf_name <- as.character(settings_variable_params_y_transf_name)
}
```

The `settings_variable_params_y_transf_name` variable name is given by the union of the name of the section containing the parameter and the name of the parameter itself. Finally, there are two pieces of code at the end of the script. One is used to remove some of the icons in the Plotly Modebar:

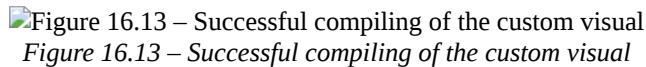
```
disabledButtonsList <- list(
  'toImage', 'sendDataToCloud', 'zoom2d', 'pan',
  'pan2d', 'select2d', 'lasso2d',
  'hoverClosestCartesian', 'hoverCompareCartesian')
p$x$config$modeBarButtonsToRemove = disabledButtonsList
p <- config(p, staticPlot = FALSE, editable = FALSE, sendData = FALSE, showLink = FALSE, dis|
```

The other is a workaround for a Plotly bug that displays the outliers of a boxplot despite passing the `outlier.shape = NA` parameter to `geom_boxplot()`:

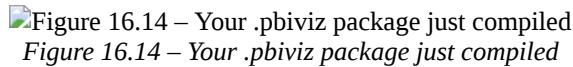
```
hideOutliers <- function(x) {
  if (x$hoverinfo == 'y') {
    x$marker = list(opacity = 0)
    x$hoverinfo = NA
  }
  return(x)
}
p[["x"]][["data"]] <- purrr::map(p[["x"]][["data"]], ~ hideOutliers(.))
```

Finally, the `internalSaveWidget(p, 'out.html')` command uses one of the utility functions loaded at the beginning of the script to generate the flattened visual in a self-contained HTML with the standard name `out.html` properly managed by Power BI. The latest command invokes the `ReadFullFileReplaceString()` function. It allows you to replace strings inside the `out.html` file generated by the code in order to modify the default configurations generated by Plotly. Specifically, the command used here corrects a setting on the padding of the generated HTML widget.

13. Now go back to the Windows PowerShell console and make sure you are in the `Power-BI-Custom-Visuals\interactiveboxplots` folder. If you were in the `Power-BI-Custom-Visuals` folder, just use the `cd interactiveboxplots` command. Then, enter the `pbviz` package command to compile the `.pbviz` package containing your custom visual. At the end of the compiling operations of `pbviz` tools, you will see something like this:



Very nice job! You compiled your very first R HTML custom visual using the `pbviz` tools. Okay, but where is the compiled package? Don't worry, look inside the `dist` folder of your Power BI Visual Project:



There it is! Let's now import it into Power BI.

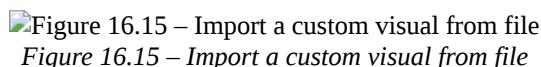
Importing the custom visual package into Power BI

Now that the bulk of the work is done, importing your custom visual into Power BI is a breeze. First of all, you need to install the `xml2` package in your R engine, as it is used by the provided utility functions:

1. Open RStudio and make sure it is referencing your latest CRAN R (version 4.0.2 in our case).
2. Click on the **Console** window and enter this command: `install.packages('xml2')`. If you remember, this library is listed in the dependency file you saw in the previous section. Then, press *Enter*.

Let's now import the custom visual in Power BI:

1. Make sure that Power BI Desktop references the correct R engine (the latest one) in the **Options**.
2. Click on **Get Data**, search for `web`, select **Web**, and click on **Connect**.
3. Enter the following URL as source: <http://bit.ly/titanic-dataset-csv>. Then press **OK**.
4. Make sure that the **File Origin** is **65001: Unicode (UTF-8)** and press **Load**.
5. Click the ellipses under the **Visuals** pane, then click on **Import a visual from a file**:



6. In the next open windows, move to the following folder:
C:\Users\<your-username>\Power-BI-Custom-Visuals\interactiveboxplots\dist . Then select your .pbviz package and click on **Open**. Click **OK** on the next dialog box.

7. As you can see, a new icon has appeared on the **Visuals** pane:

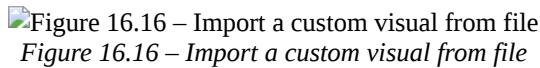


Figure 16.16 – Import a custom visual from file
Figure 16.16 – Import a custom visual from file

Keep in mind that if you want to use a custom icon for your visual, just replace it with the `icon.png` file you find in the `assets` folder of your Power BI Visual Project before compiling. Click on it to add your custom visual to your report canvas. Then click on **Enable** on the next dialog window.

8. Enlarge your custom visual area, then expand the **titanic-dataset-csv** table on the **Fields** pane and check first the **Pclass** field, then the **Fare** field:

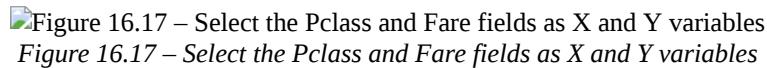


Figure 16.17 – Select the Pclass and Fare fields as X and Y variables
Figure 16.17 – Select the Pclass and Fare fields as X and Y variables

Take a look at your custom visual. You will see something like this:

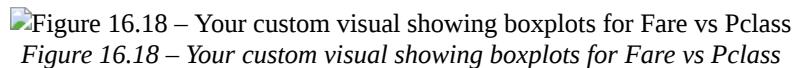


Figure 16.18 – Your custom visual showing boxplots for Fare vs Pclass
Figure 16.18 – Your custom visual showing boxplots for Fare vs Pclass

9. Now click on the **Format** icon of the visual, expand the **Variables Settings** section, and select **Yeo-Johnson** for **Y Transformation Type**:



Figure 16.19 – Select Yeo-Johnson for Y Transformation Type
Figure 16.19 – Select Yeo-Johnson for Y Transformation Type

Take now a look at your custom visual. You will see something like this:

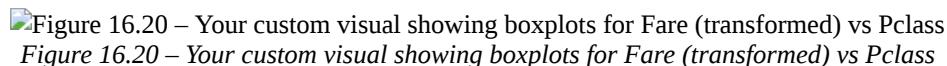


Figure 16.20 – Your custom visual showing boxplots for Fare (transformed) vs Pclass
Figure 16.20 – Your custom visual showing boxplots for Fare (transformed) vs Pclass

10. Now go back to the **titanic-dataset-csv** table on the **Fields** pane, check the **Sex** field (it will be associated to the visual's **Grouping Variable**), and take a look at the visual again. It will be like the following:

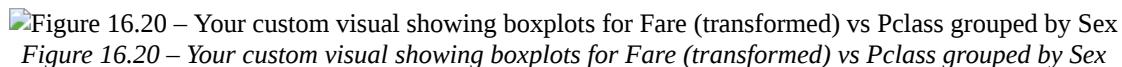


Figure 16.20 – Your custom visual showing boxplots for Fare (transformed) vs Pclass grouped by Sex
Figure 16.20 – Your custom visual showing boxplots for Fare (transformed) vs Pclass grouped by Sex

Really impressive! Your custom interactive visual is awesome and works great!

Summary

In this chapter, we learned the advantages of using an interactive visual in comparison to a static visual in some cases. We learned how to add some basic interactivity to charts developed with Ggplot via Plotly.

We learned that the key to making interactive visuals on Power BI is that they are based on HTML widgets. We have been therefore guided step by step in the realization of a custom visual compiled through the pbviz tools.

Finally, we imported the compiled package into Power BI to test its functionality. With this, we've come to the end of the book. I hope this journey was fruitful and rewarding!

References

For additional reading, check out the following books and articles:

1. *Plotly R Open Source Graphing Library* (<https://plotly.com/r/>)
2. [Course] *R: Interactive Visualizations with htmlwidgets* (<https://www.linkedin.com/learning/r-interactive-visualizations-with-htmlwidgets/welcome>)
3. *Creating a widget* (http://www.htmlwidgets.org/develop_intro.html)
4. *Power BI Visual Project Structure* (<https://docs.microsoft.com/en-us/power-bi/developer/visuals/visual-project-structure>)
5. *Schema used in the pbviz.json file* (<https://github.com/microsoft/PowerBI-visuals-tools/blob/main/templates/visuals/.api/v1.13.0/schema.pbviz.json>)
6. *Schema used in the capabilities.json file* (<https://github.com/microsoft/PowerBI-visuals-tools/blob/main/templates/visuals/.api/v1.13.0/schema.capabilities.json>)
7. *Power BI Custom Visual Part 5 – Formatting* (<https://shetland.azurewebsites.net/2021/02/18/power-bi-custom-visual-part-5-formatting/>)