

EXPERT INSIGHT



Python for Finance Cookbook

Over 80 powerful recipes for effective
financial data analysis

Second Edition



<packt>

Eryk Lewinson

Python for Finance Cookbook

Second Edition

Over 80 powerful recipes for effective financial data analysis

Eryk Lewinson



BIRMINGHAM—MUMBAI

Python for Finance Cookbook

Second Edition

Copyright © 2022 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the 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.

Senior Publishing Product Manager: Devika Battike

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Rianna Rodrigues

Content Development Editor: Edward Doxey

Copy Editor: Safis Editing

Technical Editor: Karan Sonawane

Proofreader: Safis Editing

Indexer: Rekha Nair

Presentation Designer: Ganesh Bhadwalkar

Developer Relations Marketing Executive: Vidhi Vashisth

First published: Jan 2020

Second edition: Dec 2022

Production reference: 1231222

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80324-319-1

www.packt.com

Contributors

About the author

Eryk Lewinson received his master's degree in Quantitative Finance from Erasmus University Rotterdam. In his professional career, he has gained experience in the practical application of data science methods while working in risk management and data science departments of two "big 4" companies, a Dutch neo-broker and most recently the Netherlands' largest online retailer.

Outside of work, he has written over a hundred articles about topics related to data science, which have been viewed more than 3 million times. In his free time, he enjoys playing video games, reading books, and traveling with his girlfriend.

Writing the second edition of my book was a unique experience. On the one hand, I knew what I should expect. On the other, it proved to be much more challenging in terms of both improving the existing content and expanding upon it. I must also admit that it was a very rewarding feeling to be contacted by readers with kind words about the first edition and valuable feedback on what to add and improve. Thanks to all of that, I have certainly learned a lot, and—in the end—I will remember those times fondly.

I would like to thank Agnieszka for her undeterred support and patience, my brother for once again being my first reader, and my mom for always having my back. I also greatly appreciated all the words of encouragement from my friends and colleagues. Without all of you, completing this book would not have been possible. Thank you.

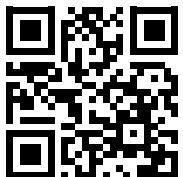
About the reviewer

Roman Paolucci is a quantitative researcher specializing in the use of applied natural language processing and machine learning to extract equity trading signals. He is the course director and founder of *Quant Guild* (<https://quantguild.com>), an online community dedicated to education on topics pertaining to quantitative finance, data science, and software engineering. Roman is also the maintainer and sole contributor of the popular quantitative finance Python library QFin, available on GitHub and PyPi for use in the simulation of stochastic processes and various derivative pricing settings. His current research interests include natural language processing, machine learning for derivative pricing, randomized numerical linear algebra, and optimal portfolio hedging via reinforcement learning.

Thank you to my family and friends—without them none of my work would be possible.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

Table of Contents

Preface	xi
<hr/>	
Chapter 1: Acquiring Financial Data	1
Getting data from Yahoo Finance	2
Getting data from Nasdaq Data Link	5
Getting data from Intrinio	9
Getting data from Alpha Vantage	16
Getting data from CoinGecko	21
Summary	24
<hr/>	
Chapter 2: Data Preprocessing	25
Converting prices to returns	25
Adjusting the returns for inflation	28
Changing the frequency of time series data	31
Different ways of imputing missing data	34
Converting currencies	39
Different ways of aggregating trade data	42
Summary	49
<hr/>	
Chapter 3: Visualizing Financial Time Series	51
Basic visualization of time series data	52
Visualizing seasonal patterns	58
Creating interactive visualizations	64
Creating a candlestick chart	69
Summary	75

Chapter 4: Exploring Financial Time Series Data	77
Outlier detection using rolling statistics	77
Outlier detection with the Hampel filter	81
Detecting changepoints in time series	86
Detecting trends in time series	92
Detecting patterns in a time series using the Hurst exponent	94
Investigating stylized facts of asset returns	98
Summary	112
Chapter 5: Technical Analysis and Building Interactive Dashboards	113
Calculating the most popular technical indicators	113
Downloading the technical indicators	120
Recognizing candlestick patterns	124
Building an interactive web app for technical analysis using Streamlit	129
Deploying the technical analysis app	139
Summary	142
Chapter 6: Time Series Analysis and Forecasting	143
Time series decomposition	144
Testing for stationarity in time series	152
Correcting for stationarity in time series	158
Modeling time series with exponential smoothing methods	166
Modeling time series with ARIMA class models	176
Finding the best-fitting ARIMA model with auto-ARIMA	190
Summary	203
Chapter 7: Machine Learning-Based Approaches to Time Series Forecasting	205
Validation methods for time series	206
Feature engineering for time series	220
Time series forecasting as reduced regression	235
Forecasting with Meta's Prophet	248
AutoML for time series forecasting with PyCaret	262
Summary	272
Chapter 8: Multi-Factor Models	275
Estimating the CAPM	276
Estimating the Fama-French three-factor model	283
Estimating the rolling three-factor model on a portfolio of assets	288

Estimating the four- and five-factor models	292
Estimating cross-sectional factor models using the Fama-MacBeth regression	298
Summary	303
Chapter 9: Modeling Volatility with GARCH Class Models	305
Modeling stock returns' volatility with ARCH models	306
Modeling stock returns' volatility with GARCH models	312
Forecasting volatility using GARCH models	316
Multivariate volatility forecasting with the CCC-GARCH model	325
Forecasting the conditional covariance matrix using DCC-GARCH	330
Summary	338
Chapter 10: Monte Carlo Simulations in Finance	339
Simulating stock price dynamics using a geometric Brownian motion	340
Pricing European options using simulations	348
Pricing American options with Least Squares Monte Carlo	353
Pricing American options using QuantLib	357
Pricing barrier options	361
Estimating Value-at-Risk using Monte Carlo	363
Summary	369
Chapter 11: Asset Allocation	371
Evaluating an equally-weighted portfolio's performance	372
Finding the efficient frontier using Monte Carlo simulations	381
Finding the efficient frontier using optimization with SciPy	389
Finding the efficient frontier using convex optimization with CVXPY	397
Finding the optimal portfolio with Hierarchical Risk Parity	405
Summary	413
Chapter 12: Backtesting Trading Strategies	415
Vectorized backtesting with pandas	417
Event-driven backtesting with backtrader	424
Backtesting a long/short strategy based on the RSI	433
Backtesting a buy/sell strategy based on Bollinger bands	440
Backtesting a moving average crossover strategy using crypto data	447
Backtesting a mean-variance portfolio optimization	454
Summary	459

Chapter 13: Applied Machine Learning: Identifying Credit Default	461
Loading data and managing data types	462
Exploratory data analysis	470
Splitting data into training and test sets	488
Identifying and dealing with missing values	492
Encoding categorical variables	499
Fitting a decision tree classifier	505
Organizing the project with pipelines	519
Tuning hyperparameters using grid searches and cross-validation	529
Summary	540
Chapter 14: Advanced Concepts for Machine Learning Projects	543
Exploring ensemble classifiers	544
Exploring alternative approaches to encoding categorical features	553
Investigating different approaches to handling imbalanced data	562
Leveraging the wisdom of the crowds with stacked ensembles	573
Bayesian hyperparameter optimization	580
Investigating feature importance	597
Exploring feature selection techniques	610
Exploring explainable AI techniques	623
Summary	643
Chapter 15: Deep Learning in Finance	645
Exploring fastai's Tabular Learner	646
Exploring Google's TabNet	658
Time series forecasting with Amazon's DeepAR	669
Time series forecasting with NeuralProphet	682
Summary	699
Other Books You May Enjoy	705
Index	709

Preface

In the last few years, we have seen spectacular growth in the field of data science. Almost every day there is some kind of new development, for example, a research paper announcing a new or improved machine learning or deep learning algorithm, or a new library for one of the most popular programming languages.

In the past, many of those advances did not make it to mainstream media. But that is also changing rapidly. Some of the recent examples include the AlphaGo program beating the 18-time world champion at Go, using deep learning to generate realistic faces of humans that never existed, or the beautiful digital art created from a text caption using models such as DALL-E 2 or Stable Diffusion.

Another example of recent and spectacular development is OpenAI's ChatGPT. It is a language model with which we can engage in natural-sounding conversations. The model is able to keep track of past questions and follow up on them, admit its mistakes, or reject inappropriate requests. What is more, it is not only restricted to natural language, we can ask it to write actual code snippets in various programming languages.

Aside from those newsworthy achievements, in the last decades AI has been adopted in virtually every industry. We can see it all around us, for example, the recommendations we get on Netflix or the emails we receive about an extra discount from an online shop that we have not used recently. As such, businesses all over the world employ AI to gain a competitive edge in the following ways:

- Making better, data-driven decisions
- Increasing their profits by efficient targeting or spot-on recommendations
- Reducing customer churn by early identification of customers at risk
- Automating repetitive tasks that AI can complete much faster (and potentially more accurately) than a human employee

The very same AI revolution is affecting the financial industry. In a 2020 article, Forbes reported that “70% of all financial services firms are using machine learning to predict cash flow events, fine-tune credit scores and detect fraud”. Additionally, various aspects of data science are also used for algorithmic trading, robo-advisory services, personalized banking, process automation, and more.

This book presents a recipe-based guide on how to solve various tasks within the financial domain using modern Python libraries. As such, we try to reduce the amount of code that needs to be written by leveraging mature and “battle-tested” libraries used by professionals in many industries. While the book assumes some prior knowledge and does not explain all the concepts from the theoretical point of view, it provides relevant references that allow the readers to dive deeper into the topics.

In this preface, you will find an outline of what you can expect from the book, how the content is organized, and what you need to meet your goals while having hands-on fun on the way. I hope you will enjoy it!

Who this book is for

This book is intended for data analysts, financial analysts, data scientists, or ML engineers who want to learn how to implement a broad range of tasks in a financial context. The book assumes that the readers have some understanding of financial markets and trading strategies. They should also be comfortable with using Python and its popular libraries oriented towards data science (for example, `pandas`, `numpy`, and `scikit-learn`).

The book will help readers to correctly use advanced approaches to data analysis within the financial domain, avoid potential pitfalls and common mistakes, and reach correct conclusions for the problems they might be trying to solve. Additionally, as the data science and financial fields are dynamically changing and expanding, the book contains references to academic papers and other relevant resources to broaden the understanding of the covered topics.

What this book covers

Chapter 1, Acquiring Financial Data, covers a few of the most popular sources of high-quality financial data, including Yahoo Finance, Nasdaq Data Link, Intrinio, and Alpha Vantage. It focuses on leveraging dedicated Python libraries and processing data for further analysis.

Chapter 2, Data Preprocessing, describes various techniques used to preprocess data. It describes the crucial steps between obtaining the data and using it for building machine learning models or investigating trading strategies. As such, it covers topics such as converting prices to returns, adjusting them for inflation, imputing missing values, or aggregating trade data into various kinds of bars.

Chapter 3, Visualizing Financial Time Series, focuses on visualizing financial (and not only) time series data. By plotting the data, we can visually identify some patterns, such as trends, seasonality, and changepoints, which we can further confirm using statistical tests. The insights gathered at this point can lead to making better decisions while choosing the modeling approach.

Chapter 4, Exploring Financial Time Series Data, shows how to use various algorithms and statistical tests to automatically identify potential issues with time series data, such as the existence of outliers. Additionally, it covers analyzing data for the existence of trends or other patterns such as mean-reversion. Lastly, it explores the stylized facts of asset returns. Together, those concepts are crucial while working with financial data, as we want to make sure that the models/strategies we are building can accurately capture the dynamics of asset returns.

Chapter 5, Technical Analysis and Building Interactive Dashboards, explains the basics of technical analysis in Python by showing how to calculate some of the most popular indicators and automatically recognize patterns in candlestick data. It also demonstrates how to create a Streamlit-based web app, which enables us to visualize and inspect the predefined TA indicators in an interactive fashion.

Chapter 6, Time Series Analysis and Forecasting, introduces the basics of time series modeling. It starts by looking into the building blocks of time series and how to separate them using various decomposition methods. Then, it covers the concept of stationarity, how to test for it, and how to achieve it in case the original series is not stationary. Lastly, it shows how to use two of the most widely used statistical approaches to time series modeling—the exponential smoothing methods and ARIMA class models.

Chapter 7, Machine Learning-Based Approaches to Time Series Forecasting, starts by explaining different ways of validating time series models. Then, it provides an overview of feature engineering approaches. It also introduces a tool for automatic feature extraction which generates hundreds or thousands of features with a few lines of code. Furthermore, it explains the concept of reduced regression and how to use Meta's popular Prophet algorithm. The chapter concludes with an introduction to one of the popular AutoML frameworks for time series forecasting.

Chapter 8, Multi-Factor Models, covers estimating various factor models, starting with the simplest one-factor model (CAPM) and then extending it to the more advanced three-, four-, and five-factor models.

Chapter 9, Modeling Volatility with GARCH Class Models, focuses on volatility and the concept of conditional heteroskedasticity. It shows how to use univariate and multivariate GARCH models, which are one of the most popular ways of modeling and forecasting volatility.

Chapter 10, Monte Carlo Simulations in Finance, explains how to use Monte Carlo methods for various tasks, such as simulating stock prices, pricing derivatives with no closed-form solution (American/Exotic options), or estimating the uncertainty of a portfolio (for example, by calculating Value-at-Risk and Expected Shortfall).

Chapter 11, Asset Allocation, starts by explaining the most basic asset allocation strategy, and on its basis, showing how to evaluate the performance of portfolios. Then it shows three different approaches to obtaining the efficient frontier. Lastly, it explores Hierarchical Risk Parity, which is a novel approach to asset allocation based on the combination of graph theory and machine learning.

Chapter 12, Backtesting Trading Strategies, presents how to run backtests of various trading strategies using two approaches (vectorized and event-driven) with the help of popular Python libraries. To do so, it uses a few examples of strategies built on the basis of popular technical indicators or mean-variance portfolio optimization.

Chapter 13, Applied Machine Learning: Identifying Credit Default, shows how to approach a real-life machine learning task of predicting loan defaults. It covers the entire scope of a machine learning project, from gathering and cleaning data to building and tuning a classifier. An important takeaway from this chapter is understanding the general approach to machine learning projects, which can then be applied to many different tasks, be it churn prediction or estimating the price of new real estate in a neighborhood.

Chapter 14, Advanced Concepts for Machine Learning Projects, continues from the workflow introduced in the preceding chapter and demonstrates possible extensions to the MVP stage of ML projects. It starts with presenting more advanced classifiers. Then, it covers alternative approaches to encoding categorical features and describes a few methods of dealing with imbalanced data.

Furthermore, it shows how to create stacked ensembles of ML models and leverage Bayesian hyperparameter tuning to improve upon exhaustive grid search. It also explores various approaches to calculating feature importance and using it to select the most informative predictors. Lastly, it touches upon the rapidly developing field of explainable AI.

Chapter 15, Deep Learning in Finance, describes how to apply some of the recent neural network architectures to two possible use cases in the financial domain—predicting credit card default (a classification task) and forecasting time series.

To get the most out of this book

In this book, we attempt to give the readers a high-level overview of various techniques used in the financial domain, while focusing on the practical applications of these methods. That is why we put special emphasis on showing how to use various popular Python libraries to make the work of an analyst or data scientist much easier and less prone to errors.

As the best way to learn anything is by doing, we highly encourage the readers to experiment with the code samples provided (the code can be found in the accompanying GitHub repository), apply the techniques to different datasets, and explore possible extensions (some of them mentioned in the *See also* sections of the recipes).

For a deeper dive into the theoretical foundations, we provide references for further reading. Those also include even more advanced techniques that are outside of the scope of this book.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Python-for-Finance-Cookbook-2E>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/JnpTe>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, names of Python libraries, database table names, folder names, filenames, file extensions, and pathnames. For example: “We can also use the `get_by_id` function to download a particular CPI series.”

A block of code is set as follows:

```
def realized_volatility(x):
    return np.sqrt(np.sum(x**2))
```

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

```
Downloaded 2769 rows of data.
```

Bold: Indicates a new term or an important word. For example: “**Volume bars** are an attempt at overcoming this problem “



Information boxes appear like this.



Tips and tricks appear like this.

Furthermore, at the very beginning of each Jupyter Notebook (available on the book’s GitHub repository), we run a few cells that import and set up plotting with `matplotlib`. For brevity’s sake, we will not mention this later on in the book. So at any time, assume that the following commands were executed.

First, we (optionally) increased the resolution of the generated figures using the following snippet:

```
%config InlineBackend.figure_format = "retina"
```

Then we execute the second snippet:

```
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
from pandas.core.common import SettingWithCopyWarning
warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=SettingWithCopyWarning)

# feel free to modify, for example, change the context to "notebook"
sns.set_theme(context="talk", style="whitegrid",
               palette="colorblind", color_codes=True,
               rc={"figure.figsize": [12, 8]})
```

In this cell, we import `matplotlib`, `warnings`, and `seaborn`. Then, we disabled some of the warnings and set up the style of the plots. In some chapters, we might modify these settings for better readability of the figures (especially in black and white).

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

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

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

Share your thoughts

Once you've read *Python for Finance Cookbook - Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781803243191>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Acquiring Financial Data

The first chapter of this book is dedicated to a very important (some may say the most important) part of any data science/quantitative finance project—gathering data. In line with the famous adage “garbage in, garbage out,” we should strive to obtain data of the highest possible quality and then correctly preprocess it for later use with statistical and machine learning algorithms. The reason for this is simple—the results of our analyses are highly dependent on the input data and no sophisticated model will be able to compensate for that. That is also why in our analyses, we should be able to use our (or someone else’s) understanding of the economic/financial domain to motivate certain data for, for example, modeling stock returns.

One of the most frequently reported issues among the readers of the first edition of this book was getting high-quality data. That is why in this chapter we spend more time exploring different sources of financial data. While quite a few of these vendors offer similar information (prices, fundamentals, and so on), they also offer additional, unique data that can be downloaded via their APIs. An example could be company-related news articles or pre-computed technical indicators. That is why we will download different types of data depending on the recipe. However, be sure to inspect the documentation of the library/API, as most likely its vendor also provides standard data such as prices.



Additional examples are also covered in the Jupyter notebooks, which you can find in the accompanying GitHub repository.

The data sources in this chapter were selected intentionally not only to showcase how easy it can be to gather high-quality data using Python libraries but also to show that the gathered data comes in many shapes and sizes.

Sometimes we will get a nicely formatted pandas DataFrame, while other times it might be in JSON format or even bytes that need to be processed and then loaded as a CSV. Hopefully, these recipes will sufficiently prepare you to work with any kind of data you might encounter online.

Something to bear in mind while reading this chapter is that data differs among sources. This means that the prices we downloaded from two vendors will most likely differ, as those vendors also get their data from different sources and might use other methods to adjust the prices for corporate actions. The best practice is to find a source you trust the most concerning a particular type of data (based on, for example, opinion on the internet) and then use it to download the data you need. One additional thing to keep in mind is that when building algorithmic trading strategies, the data we use for modeling should align with the live data feed used for executing the trades.

This chapter does not cover one important type of data—alternative data. This could be any type of data that can be used to generate some insights into predicting asset prices. Alternative data can include satellite images (for example, tracking shipping routes, or the development of a certain area), sensor data, web traffic data, customer reviews, etc. While there are many vendors specializing in alternative data (for example, Quandl/Nasdaq Data Link), you can also get some by accessing publicly available information via web scraping. As an example, you could scrape customer reviews from Amazon or Yelp. However, those are often bigger projects and are unfortunately outside of the scope of this book. Also, you need to make sure that web scraping a particular website is not against its terms and conditions!



Using the vendors mentioned in this chapter, you can get quite a lot of information for free. But most of those providers also offer paid tiers. Remember to do thorough research on what the data suppliers actually provide and what your needs are before signing up for any of the services.

In this chapter, we cover the following recipes:

- Getting data from Yahoo Finance
- Getting data from Nasdaq Data Link
- Getting data from Intrinio
- Getting data from Alpha Vantage
- Getting data from CoinGecko

Getting data from Yahoo Finance

One of the most popular sources of free financial data is **Yahoo Finance**. It contains not only historical and current stock prices in different frequencies (daily, weekly, and monthly), but also calculated metrics, such as the **beta** (a measure of the volatility of an individual asset in comparison to the volatility of the entire market), fundamentals, earnings information/calendars, and many more.



For a long period of time, the go-to tool for downloading data from Yahoo Finance was the `pandas-datareader` library. The goal of the library was to extract data from a variety of sources and store it in the form of a `pandas DataFrame`. However, after some changes to the Yahoo Finance API, this functionality was deprecated. It is definitely good to be familiar with this library, as it facilitates downloading data from sources such as FRED (Federal Reserve Economic Data), the Fama/French Data Library, or the World Bank. Those might come in handy for different kinds of analyses and some of them are presented in the following chapters.

As of now, the easiest and fastest way of downloading historical stock prices is to use the `yfinance` library (formerly known as `fix_yahoo_finance`).

For the sake of this recipe, we are interested in downloading Apple's stock prices from the years 2011 to 2021.

How to do it...

Execute the following steps to download data from Yahoo Finance:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
```

2. Download the data:

```
df = yf.download("AAPL",
                  start="2011-01-01",
                  end="2021-12-31",
                  progress=False)
```

3. Inspect the downloaded data:

```
print(f"Downloaded {len(df)} rows of data.")
df
```

Running the code generates the following preview of the DataFrame:

Downloaded 2769 rows of data.

	Open	High	Low	Close	Adj Close	Volume
Date						
2010-12-31	11.533929	11.552857	11.475357	11.520000	9.864279	193508000
2011-01-03	11.630000	11.795000	11.601429	11.770357	10.078650	445138400
2011-01-04	11.872857	11.875000	11.719643	11.831786	10.131254	309080800
2011-01-05	11.769643	11.940714	11.767857	11.928571	10.214125	255519600
2011-01-06	11.954286	11.973214	11.889286	11.918929	10.205874	300428800
...
2021-12-23	175.850006	176.850006	175.270004	176.279999	176.055695	68356600
2021-12-27	177.089996	180.419998	177.070007	180.330002	180.100540	74919600
2021-12-28	180.160004	181.330002	178.529999	179.289993	179.061859	79144300
2021-12-29	179.330002	180.630005	178.139999	179.380005	179.151749	62348900
2021-12-30	179.470001	180.570007	178.089996	178.199997	177.973251	59773000

Figure 1.1: Preview of the DataFrame with downloaded stock prices

The result of the request is a pandas DataFrame (2,769 rows) containing daily Open, High, Low, and Close (OHLC) prices, as well as the adjusted close price and volume.

Yahoo Finance automatically adjusts the close price for **stock splits**, that is, when a company divides the existing shares of its stock into multiple new shares, most frequently to boost the stock's liquidity. The adjusted close price takes into account not only splits but also dividends.

How it works...

The `download` function is very intuitive. In the most basic case, we just need to provide the ticker (symbol), and it will try to download all available data since 1950.

In the preceding example, we downloaded daily data from a specific range (2011 to 2021).

Some additional features of the `download` function are:

- We can download information for multiple tickers at once by providing a list of tickers (["AAPL", "MSFT"]) or multiple tickers as a string ("AAPL MSFT").
- We can set `auto_adjust=True` to download only the adjusted prices.
- We can additionally download dividends and stock splits by setting `actions='inline'`. Those actions can also be used to manually adjust the prices or for other analyses.
- Specifying `progress=False` disables the progress bar.
- The `interval` argument can be used to download data in different frequencies. We could also download intraday data as long as the requested period is shorter than 60 days.

There's more...

yfinance also offers an alternative way of downloading the data—via the `Ticker` class. First, we need to instantiate the object of the class:

```
aapl_data = yf.Ticker("AAPL")
```

To download the historical price data, we can use the `history` method:

```
aapl_data.history()
```

By default, the method downloads the last month of data. We can use the same arguments as in the `download` function to specify the range and frequency.

The main benefit of using the `Ticker` class is that we can download much more information than just the prices. Some of the available methods include:

- `info`—outputs a JSON object containing detailed information about the stock and its company, for example, the company's full name, a short business summary, which exchange it is listed on, as well as a selection of financial metrics such as the beta coefficient
- `actions`—outputs corporate actions such as dividends and splits
- `major_holders`—presents the names of the major holders
- `institutional_holders`—shows the institutional holders

- `calendar`—shows the incoming events, such as the quarterly earnings
- `earnings/quarterly_earnings`—shows the earnings information from the last few years/quarters
- `financials/quarterly_financials`—contains financial information such as income before tax, net income, gross profit, EBIT, and much more



Please see the corresponding Jupyter notebook for more examples and outputs of those methods.

See also

For a complete list of downloadable data, please refer to the GitHub repo of `yfinance` (<https://github.com/ranaroussi/yfinance>).

You can check out some alternative libraries for downloading data from Yahoo Finance:

- `yahoofinancials`—similarly to `yfinance`, this library offers the possibility of downloading a wide range of data from Yahoo Finance. The biggest difference is that all the downloaded data is returned as JSON.
- `yahoo_earnings_calendar`—a small library dedicated to downloading the earnings calendar.

Getting data from Nasdaq Data Link

Alternative data can be anything that is considered non-market data, for example, weather data for agricultural commodities, satellite images that track oil shipments, or even customer feedback that reflects a company's service performance. The idea behind using alternative data is to get an “informational edge” that can then be used for generating alpha. In short, `alpha` is a measure of performance describing an investment strategy’s, trader’s, or portfolio manager’s ability to beat the market.

`Quandl` was the leading provider of alternative data products for investment professionals (including quant funds and investment banks). Recently, it was acquired by Nasdaq and is now part of the `Nasdaq Data Link` service. The goal of the new platform is to provide a unified source of trusted data and analytics. It offers an easy way to download data, also via a dedicated Python library.

A good starting place for financial data would be the `WIKI` Prices database, which contains stock prices, dividends, and splits for 3,000 US publicly traded companies. The drawback of this database is that as of April 2018, it is no longer supported (meaning there is no recent data). However, for purposes of getting historical data or learning how to access the databases, it is more than enough.

We use the same example that we used in the previous recipe—we download Apple’s stock prices for the years 2011 to 2021.

Getting ready

Before downloading the data, we need to create an account at Nasdaq Data Link (<https://data.nasdaq.com/>) and then authenticate our email address (otherwise, an exception is likely to occur while downloading the data). We can find our personal API key in our profile (<https://data.nasdaq.com/account/profile>).

How to do it...

Execute the following steps to download data from Nasdaq Data Link:

1. Import the libraries:

```
import pandas as pd
import nasdaqdatalink
```

2. Authenticate using your personal API key:

```
nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

You need to replace YOUR_KEY_HERE with your own API key.

3. Download the data:

```
df = nasdaqdatalink.get(dataset="WIKI/AAPL",
                         start_date="2011-01-01",
                         end_date="2021-12-31")
```

4. Inspect the downloaded data:

```
print(f"Downloaded {len(df)} rows of data.")
df.head()
```

Running the code generates the following preview of the DataFrame:

Downloaded 1818 rows of data.														
Date	Open	High	Low	Close	Volume	Ex-Dividend	Split Ratio	Adj. Open	Adj. High	Adj. Low	Adj. Close	Adj. Volume		
2011-01-03	325.6400	330.26	324.8365	329.57	15897800.0	0.0	1.0	41.849279	42.443013	41.746018	42.354338	111284600.0		
2011-01-04	332.4400	332.50	328.1500	331.29	11038600.0	0.0	1.0	42.723173	42.730884	42.171849	42.575382	77270200.0		
2011-01-05	329.5500	334.34	329.5000	334.00	9125700.0	0.0	1.0	42.351768	42.967350	42.345342	42.923655	63879900.0		
2011-01-06	334.7194	335.25	332.9000	333.73	10729600.0	0.0	1.0	43.016108	43.084298	42.782290	42.888956	75107200.0		
2011-01-07	333.9900	336.35	331.9000	336.12	11140400.0	0.0	1.0	42.922370	43.225663	42.653776	43.196105	77982800.0		

Figure 1.2: Preview of the downloaded price information

The result of the request is a DataFrame (1,818 rows) containing the daily OHLC prices, the adjusted prices, dividends, and potential stock splits. As we mentioned in the introduction, the data is limited and is only available until April 2018—the last observation actually comes from March, 27 2018.

How it works...

The first step after importing the required libraries was authentication using the API key. When providing the dataset argument, we used the following structure: DATASET/TICKER.



We should keep the API keys secure and private, that is, not share them in public repositories, or anywhere else. One way to make sure that the key stays private is to create an environment variable (how to do it depends on your operating system) and then load it in Python. To do so, we can use the `os` module. To load the `NASDAQ_KEY` variable, we could use the following code: `os.environ.get("NASDAQ_KEY")`.

Some additional details on the `get` function are:

- We can specify multiple datasets at once using a list such as `["WIKI/AAPL", "WIKI/MSFT"]`.
- The `collapse` argument can be used to define the frequency (available options are daily, weekly, monthly, quarterly, or annually).
- The `transform` argument can be used to carry out some basic calculations on the data prior to downloading. For example, we could calculate row-on-row change (`diff`), row-on-row percentage change (`rdiff`), or cumulative sum (`cumul`) or scale the series to start at 100 (`normalize`). Naturally, we can easily do the very same operation using `pandas`.

There's more...

Nasdaq Data Link distinguishes two types of API calls for downloading data. The `get` function we used before is classified as a time-series API call. We can also use the tables API call with the `get_table` function.

1. Download the data for multiple tickers using the `get_table` function:

```
COLUMNS = ["ticker", "date", "adj_close"]
df = nasdaqlink.get_table("WIKI/PRICES",
                           ticker=["AAPL", "MSFT", "INTC"],
                           qopts={"columns": COLUMNS},
                           date={"gte": "2011-01-01",
                                 "lte": "2021-12-31"},
                           paginate=True)
df.head()
```

2. Running the code generates the following preview of the DataFrame:

	ticker	date	adj_close
None			
0	MSFT	2018-03-27	89.47
1	MSFT	2018-03-26	93.78
2	MSFT	2018-03-23	87.18
3	MSFT	2018-03-22	89.79
4	MSFT	2018-03-21	92.48

Figure 1.3: Preview of the downloaded price data

This function call is a bit more complex than the one we did with the get function. We first specified the table we want to use. Then, we provided a list of tickers. As the next step, we specified which columns of the table we were interested in. We also provided the range of dates, where gte stands for *greater than or equal to*, while lte is *less than or equal to*. Lastly, we also indicated we wanted to use pagination. The *tables* API is limited to 10,000 rows per call. However, by using `paginate=True` in the function call we extend the limit to 1,000,000 rows.

3. Pivot the data from long format to wide:

```
df = df.set_index("date")
df_wide = df.pivot(columns="ticker")
df_wide.head()
```

Running the code generates the following preview of the DataFrame:

ticker	adj_close		
	AAPL	INTC	MSFT
date			
2011-01-03	42.354338	16.488706	23.211568
2011-01-04	42.575382	16.725954	23.300747
2011-01-05	42.923655	16.559880	23.228159
2011-01-06	42.888956	16.425440	23.908412
2011-01-07	43.196105	16.338449	23.725905

Figure 1.4: Preview of the pivoted DataFrame

The output of the `get_tables` function is in the long format. However, to make our analyses easier, we might be interested in the wide format. To reshape the data, we first set the `date` column as an index and then used the `pivot` method of a `pd.DataFrame`.



Please bear in mind that this is not the only way to do so, and pandas contains at least a few helpful methods/functions that can be used for reshaping the data from long to wide and vice versa.

See also

- <https://docs.data.nasdaq.com/docs/python>—the documentation of the nasdaqdatalink library for Python.
- <https://data.nasdaq.com/publishers/zacks>—Zacks Investment Research is a provider of various financial data that might be relevant for your projects. Please bear in mind that the data is not free (you can always get a preview of the data before purchasing access).
- <https://data.nasdaq.com/publishers>—a list of all the available data providers.

Getting data from Intrinio

Another interesting source of financial data is Intrinio, which offers access to its free (with limits) database. The following list presents just a few of the interesting data points that we can download using Intrinio:

- Intraday historical data
- Real-time stock/option prices
- Financial statement data and fundamentals
- Company news
- Earnings-related information
- IPOs
- Economic data such as the Gross Domestic Product (GDP), unemployment rate, federal funds rate, etc.
- 30+ technical indicators

Most of the data is free of charge, with some limits on the frequency of calling the APIs. Only the real-time price data of US stocks and ETFs requires a different kind of subscription.

In this recipe, we follow the preceding example of downloading Apple's stock prices for the years 2011 to 2021. That is because the data returned by the API is not simply a pandas DataFrame and requires some interesting preprocessing.

Getting ready

Before downloading the data, we need to register at <https://intrinio.com> to obtain the API key.

Please see the following link (<https://docs.intrinio.com/developer-sandbox>) to understand what information is included in the sandbox API key (the free one).

How to do it...

Execute the following steps to download data from Intrinio:

1. Import the libraries:

```
import intrinio_sdk as intrinio
import pandas as pd
```

- Authenticate using your personal API key, and select the API:

```
intrinio.ApiClient().set_api_key("YOUR_KEY_HERE")
security_api = intrinio.SecurityApi()
```

You need to replace YOUR_KEY_HERE with your own API key.

- Request the data:

```
r = security_api.get_security_stock_prices(
    identifier="AAPL",
    start_date="2011-01-01",
    end_date="2021-12-31",
    frequency="daily",
    page_size=10000
)
```

- Convert the results into a DataFrame:

```
df = (
    pd.DataFrame(r.stock_prices_dict)
    .sort_values("date")
    .set_index("date")
)
```

- Inspect the data:

```
print(f"Downloaded {df.shape[0]} rows of data.")
df.head()
```

The output looks as follows:

Downloaded 2675 rows of data.													
	intraperiod	frequency	open	high	low	close	volume	adj_open	adj_high	adj_low	adj_close	adj_volume	
date													
2011-01-03	False	daily	325.90	330.26	324.8365	329.57	15897201.0	9.993683	10.127381	9.961070	10.106223	445121628.0	
2011-01-04	False	daily	332.50	332.50	328.1500	331.29	11048143.0	10.196071	10.196071	10.062679	10.158966	309348004.0	
2011-01-05	False	daily	329.55	334.34	329.5000	334.00	9125599.0	10.105609	10.252494	10.104076	10.242068	255516772.0	
2011-01-06	False	daily	335.00	335.25	332.9000	333.73	10729518.0	10.272733	10.280399	10.208337	10.233789	300426504.0	
2011-01-07	False	daily	334.12	336.35	331.9000	336.12	11140316.0	10.245748	10.314130	10.177672	10.307078	311928848.0	

Figure 1.5: Preview of the downloaded price information

The resulting DataFrame contains the OHLC prices and volume, as well as their adjusted counterparts. However, that is not all, and we had to cut out some additional columns to make the table fit the page. The DataFrame also contains information, such as split ratio, dividend, change in value, percentage change, and the 52-week rolling high and low values.

How it works...

The first step after importing the required libraries was to authenticate using the API key. Then, we selected the API we wanted to use for the recipe—in the case of stock prices, it was the `SecurityApi`.

To download the data, we used the `get_security_stock_prices` method of the `SecurityApi` class. The parameters we can specify are as follows:

- `identifier`—stock ticker or another acceptable identifier
- `start_date/end_date`—these are self-explanatory
- `frequency`—which data frequency is of interest to us (available choices: daily, weekly, monthly, quarterly, or yearly)
- `page_size`—defines the number of observations to return on one page; we set it to a high number to collect all the data we need in one request with no need for the `next_page` token

The API returns a JSON-like object. We accessed the dictionary form of the response, which we then transformed into a `DataFrame`. We also set the date as an index using the `set_index` method of a `pandas` `DataFrame`.

There's more...

In this section, we show some more interesting features of Intrinio.



Not all information is included in the free tier. For a more thorough overview of what data we can download for free, please refer to the following documentation page: <https://docs.intrinio.com/developer-sandbox>.

Get Coca-Cola's real-time stock price

You can use the previously defined `security_api` to get the real-time stock prices:

```
security_api.get_security_realtime_price("KO")
```

The output of the snippet is the following JSON:

```
{'ask_price': 57.57,
 'ask_size': 114.0,
 'bid_price': 57.0,
 'bid_size': 1.0,
 'close_price': None,
 'exchange_volume': 349353.0,
 'high_price': 57.55,
 'last_price': 57.09,
 'last_size': None,
 'last_time': datetime.datetime(2021, 7, 30, 21, 45, 38, tzinfo=tzutc()),
 'low_price': 48.13,
```

```
'market_volume': None,
'open_price': 56.91,
'security': {'composite_figi': 'BBG000BMX289',
             'exchange_ticker': 'KO:UN',
             'figi': 'BBG000BMX4N8',
             'id': 'sec_X7m9Zy',
             'ticker': 'KO'},
'source': 'bats_delayed',
'updated_on': datetime.datetime(2021, 7, 30, 22, 0, 40, 758000,
tzinfo=tzutc())}
```

Download news articles related to Coca-Cola

One of the potential ways to generate trading signals is to aggregate the market's sentiment on the given company. We could do it, for example, by analyzing news articles or tweets. If the sentiment is positive, we can go long, and vice versa. Below, we show how to download news articles about Coca-Cola:

```
r = intrinio.CompanyApi().get_company_news(
    identifier="KO",
    page_size=100
)

df = pd.DataFrame(r.news_dict)
df.head()
```

This code returns the following DataFrame:

	id	title	publication_date	url	summary
0	nws_1ExBnx	12 Best Blue-Chip Stocks Right Now	2021-08-09 20:27:39+00:00	https://finance.yahoo.com/news/12-best-blue-ch...	In this article, we will look at the 12 best b...
1	nws_JbL8mV	The Coca-Cola Company (NYSE:KO) Yields 3% With...	2021-08-09 09:26:07+00:00	https://finance.yahoo.com/news/coca-cola-compa...	The Coca-Cola Company NYSE:KO) is a staple sto...
2	nws_DkAPK0	10 High Yield Monthly Dividend Stocks to Buy i...	2021-08-07 13:57:00+00:00	https://finance.yahoo.com/news/10-high-yield-m...	In this article, we will be looking at 10 high...
3	nws_pRYkD9	10 Best Dividend Paying Stocks to Buy Now	2021-08-04 14:32:49+00:00	https://finance.yahoo.com/news/10-best-dividen...	In this article, we will be looking at the 10 ...
4	nws_kpVJDP	PepsiCo (PEP) Agrees to Offload Its Juice Bran...	2021-08-04 13:21:01+00:00	https://finance.yahoo.com/news/pepsioco-pep...	PepsiCo (PEP) unveils plans to offload juice b...

Figure 1.6: Preview of the news about the Coca-Cola company

Search for companies connected to the search phrase

Running the following snippet returns a list of companies that Intrinio's Thea AI recognized based on the provided query string:

```
r = intrinio.CompanyApi().recognize_company("Intel")
df = pd.DataFrame(r.companies_dict)
df
```

As we can see, there are quite a few companies that also contain the phrase “intel” in their names, other than the obvious search result.

	id	ticker	name			lei	cik
0	com_gPQrmX	I	Intelsat SA			None	0001525773
1	com_yRZOxy	IHSI	Intelligent Highway Solutions Inc			None	0001549719
2	com_gQQr4g	INTB	Intelligent Buying, Inc.			None	0001358633
3	com_gwk3Qg	SVFC	Intellicell Biosciences Inc			None	0001125280
4	com_NgYGzd	INTC	Intel Corp		KNX4USFCNCPY45LOCE31	0000050863	
5	com_gYnr4X	None	Inteliquest Inc		549300K8G7V0F3VFUL90	0001292653	
6	com_ybNLQy	ILNS	Intellect Neurosciences Inc			None	0001337905
7	com_y1jq9g	ITTI	INTELLECTUAL TECHNOLOGY INC			None	0000859914
8	com_XGb25g	DTFSF	Intelimax Media Inc.			None	0001434598
9	com_ybNdy	ILIV	Intelligent Living America Inc			None	0001141673

Figure 1.7: Preview of the companies connected to the phrase “intel”

Get Coca-Cola’s intraday stock prices

We can also retrieve intraday prices using the following snippet:

```
response = (
    security_api.get_security_intraday_prices(identifier="KO",
                                                start_date="2021-01-02",
                                                end_date="2021-01-05",
                                                page_size=1000)
)
df = pd.DataFrame(response.intraday_prices_dict)
df
```

Which returns the following DataFrame containing intraday price data.

	time	last_price	ask_price	ask_size	bid_price	bid_size	volume	source
0	2021-01-04 20:59:58+00:00	52.755	55.00	100.0	52.30	234.0	0.0	None
1	2021-01-04 20:59:57+00:00	52.745	55.00	100.0	52.74	200.0	870885.0	None
2	2021-01-04 20:59:54+00:00	52.745	55.00	100.0	52.30	234.0	870641.0	None
3	2021-01-04 20:59:51+00:00	52.740	55.00	100.0	52.30	234.0	870341.0	None
4	2021-01-04 20:59:49+00:00	52.725	52.73	600.0	52.71	300.0	868833.0	None
...
995	2021-01-04 19:01:07+00:00	52.490	52.50	100.0	52.48	200.0	582993.0	None
996	2021-01-04 19:01:03+00:00	52.510	52.50	200.0	52.49	200.0	582493.0	None
997	2021-01-04 19:01:02+00:00	52.495	52.51	200.0	52.49	200.0	582093.0	None
998	2021-01-04 19:00:59+00:00	52.510	52.51	200.0	52.50	200.0	580893.0	None
999	2021-01-04 19:00:51+00:00	52.505	52.51	200.0	52.50	100.0	580793.0	None

Figure 1.8: Preview of the downloaded intraday prices

Get Coca-Cola's latest earnings record

Another interesting usage of the `security_api` is to recover the latest earnings records. We can do this using the following snippet:

```
r = security_api.get_security_latest_earnings_record(identifier="KO")
print(r)
```

The output of the API call contains quite a lot of useful information. For example, we can see what time of day the earnings call happened. This information could potentially be used for implementing trading strategies that act when the market opens.

```
{'board_of_directors_meeting_date': None,
 'board_of_directors_meeting_type': None,
 'broadcast_url': 'http://mmm.wallstreethorizon.com/u.asp?u=347366',
 'company_website': 'http://mmm.wallstreethorizon.com/u.asp?u=14711',
 'conference_call_date': datetime.date(2020, 10, 22),
 'conference_call_passcode': None,
 'conference_call_phone_number': None,
 'conference_call_time': '8:30 AM',
 'last_confirmation_date': datetime.date(2020, 9, 23),
 'next_earnings_date': datetime.date(2020, 10, 22),
 'next_earnings_fiscal_year': 2020,
 'next_earnings_quarter': 'Q3',
 'preliminary_earnings_date': None,
 'q1_date': datetime.date(2020, 4, 21),
 'q2_date': datetime.date(2020, 7, 21),
 'q3_date': datetime.date(2020, 10, 22),
 'q4_date': datetime.date(2020, 1, 30),
 'quarter': 'Q3',
 'security': {'code': 'EQS',
              'company_id': 'com_VXWJgy',
              'composite_figi': 'BBG000BMX289',
              'composite_ticker': 'KO:US',
              'currency': 'USD',
              'figi': 'BBG000BMX4N8',
              'id': 'sec_X7m9Zy',
              ...
              'time_of_day': 'Before Market',
              'transcript_fiscal_year': None,
              'transcript_quarter': None,
              'transcript_url': None,
              'type': 'V'}
```

Figure 1.9: Coca-Cola's latest earnings record

3. Download the daily prices of Bitcoin, expressed in EUR:

```
data, meta_data = crypto_api.get_digital_currency_daily(
    symbol="BTC",
    market="EUR"
)
```

The `meta_data` object contains some useful information about the details of the query. You can see it below:

```
{'1. Information': 'Daily Prices and Volumes for Digital Currency',
 '2. Digital Currency Code': 'BTC',
 '3. Digital Currency Name': 'Bitcoin',
 '4. Market Code': 'EUR',
 '5. Market Name': 'Euro',
 '6. Last Refreshed': '2022-08-25 00:00:00',
 '7. Time Zone': 'UTC'}
```

The `data` DataFrame contains all the requested information. We obtained 1,000 daily OHLC prices, the volume, and the market capitalization. What is also noteworthy is that all the OHLC prices are provided in two currencies: EUR (as we requested) and USD (the default one).

date	1a. open (EUR)	1b. open (USD)	2a. high (EUR)	2b. high (USD)	3a. low (EUR)	3b. low (USD)	4a. close (EUR)	4b. close (USD)	5. volume	6. market cap (USD)
2022-02-05	36283.779760	41571.70	36480.927824	41797.58	36084.152944	41342.98	36411.034000	41717.50	2595.290800	2595.290800
2022-02-04	32565.896144	37311.98	36458.899624	41772.33	32318.929944	37026.73	36286.005400	41574.25	64703.958740	64703.958740
2022-02-03	32203.151736	36896.37	32631.373600	37387.00	31639.000000	36250.00	32565.573208	37311.61	32081.09990	32081.09990
2022-02-02	33772.638152	38694.59	33913.446976	38855.92	31933.089960	36586.95	32203.143008	36896.36	35794.681300	35794.681300
2022-02-01	33573.910320	38466.90	34270.666560	39265.20	33166.400000	38000.00	33772.638152	38694.59	34574.446630	34574.446630
...
2019-05-17	6867.775176	7868.67	6916.940000	7925.00	6033.666400	6913.00	6419.670928	7355.26	88752.008159	88752.008159
2019-05-16	7129.973024	8169.08	7261.960000	8320.00	6724.924000	7705.00	6865.959752	7866.59	69630.513996	69630.513996
2019-05-15	6934.622928	7945.26	7199.727200	8249.00	6851.480000	7850.00	7130.662536	8169.87	37884.327211	37884.327211
2019-05-14	6804.017136	7795.62	7301.844800	8366.00	6632.895968	7599.56	6936.630368	7947.56	76583.722603	76583.722603
2019-05-13	6081.879872	6968.24	7069.680000	8100.00	5996.136000	6870.00	6799.731688	7790.71	85804.735333	85804.735333

Figure 1.10: Preview of the downloaded prices, volume, and market cap

4. Download the real-time exchange rate:

```
crypto_api.get_digital_currency_exchange_rate(
    from_currency="BTC",
    to_currency="USD"
)[0].transpose()
```

Running the command returns the following DataFrame with the current exchange rate:

Realtime Currency Exchange Rate	
1. From_Currency Code	BTC
2. From_Currency Name	Bitcoin
3. To_Currency Code	USD
4. To_Currency Name	United States Dollar
5. Exchange Rate	41487.32000000
6. Last Refreshed	2022-02-05 10:51:02
7. Time Zone	UTC
8. Bid Price	41487.32000000
9. Ask Price	41487.33000000

Figure 1.11: BTC-USD exchange rate

How it works...

After importing the `alpha_vantage` library, we had to authenticate using the personal API key. We did so while instantiating an object of the `CryptoCurrencies` class. At the same time, we specified that we would like to obtain output in the form of a pandas DataFrame. The other possibilities are JSON and CSV.

In *Step 3*, we downloaded the daily BTC prices using the `get_digital_currency_daily` method. Additionally, we specified that we wanted to get the prices in EUR. By default, the method will return the requested EUR prices, as well as their USD equivalents.

Lastly, we downloaded the real-time BTC/USD exchange rate using the `get_digital_currency_exchange_rate` method.

There's more...

So far, we have used the `alpha_vantage` library as a middleman to download information from Alpha Vantage. However, the functionalities of the data vendor evolve faster than the third-party library and it might be interesting to learn an alternative way of accessing their API.

1. Import the libraries:

```
import requests
import pandas as pd
from io import BytesIO
```

2. Download Bitcoin's intraday data:

```
AV_API_URL = "https://www.alphavantage.co/query"
parameters = {
    "function": "CRYPTO_INTRADAY",
    "symbol": "ETH",
    "market": "USD",
```

```

    "interval": "30min",
    "outputsize": "full",
    "apikey": ALPHA_VANTAGE_API_KEY
}
r = requests.get(AV_API_URL, params=parameters)
data = r.json()
df = (
    pd.DataFrame(data["Time Series Crypto (30min)"])
    .transpose()
)
df

```

Running the snippet above returns the following preview of the downloaded DataFrame:

	1. open	2. high	3. low	4. close	5. volume
2022-02-05 10:30:00	3013.70000	3023.98000	3012.70000	3020.27000	2699
2022-02-05 10:00:00	3027.75000	3035.69000	3007.00000	3013.69000	7016
2022-02-05 09:30:00	3018.65000	3033.00000	3002.27000	3027.75000	12938
2022-02-05 09:00:00	3007.94000	3023.68000	3003.17000	3018.64000	4540
2022-02-05 08:30:00	3007.14000	3018.54000	2988.07000	3007.93000	7162
...
2022-01-15 17:00:00	3358.60000	3361.57000	3346.76000	3353.08000	3772
2022-01-15 16:30:00	3357.25000	3365.01000	3356.39000	3358.60000	3042
2022-01-15 16:00:00	3361.97000	3371.64000	3353.00000	3357.24000	7244
2022-01-15 15:30:00	3342.60000	3369.94000	3340.93000	3361.97000	17695
2022-01-15 15:00:00	3338.71000	3347.53000	3332.51000	3342.59000	4291

Figure 1.12: Preview of the DataFrame containing Bitcoin's intraday prices

We first defined the base URL used for requesting information via Alpha Vantage's API. Then, we defined a dictionary containing the additional parameters of the request, including the personal API key. In our function call, we specified that we want to download intraday ETH prices expressed in USD and sampled every 30 minutes. We also indicated we want a full output (by specifying the `outputsize` parameter). The other option is `compact` output, which downloads the 100 most recent observations.

Having prepared the request's parameters, we used the `get` function from the `requests` library. We provide the base URL and the `parameters` dictionary as arguments. After obtaining the response to the request, we can access it in JSON format using the `json` method. Lastly, we convert the element of interest into a pandas DataFrame.



Alpha Vantage's documentation shows a slightly different approach to downloading this data, that is, by creating a long URL with all the parameters specified there. Naturally, that is also a possibility, however, the option presented above is a bit neater. To see the very same request URL as presented by the documentation, you can run `r.request.url`.

3. Download the upcoming earnings announcements within the next three months:

```
AV_API_URL = "https://www.alphavantage.co/query"
parameters = {
    "function": "EARNINGS_CALENDAR",
    "horizon": "3month",
    "apikey": ALPHA_VANTAGE_API_KEY
}

r = requests.get(AV_API_URL, params=parameters)
pd.read_csv(BytesIO(r.content))
```

Running the snippet returns the following output:

	symbol		name	reportDate	fiscalDateEnding	estimate	currency
0	A	Agilent Technologies Inc	2022-02-22	2022-01-31	1.18	USD	
1	AA	Alcoa Corp	2022-04-13	2022-03-31	2.35	USD	
2	AACG	ATA Creativity Global	2022-03-28	2021-12-31	-0.27	USD	
3	AADI	Aadi Bioscience Inc	2022-03-09	2021-12-31	-0.98	USD	
4	AAIC	Arlington Asset Investment Corp - Class A	2022-02-14	2021-12-31	0.06	USD	
...	
7154	ZYME	Zymeworks Inc	2022-02-22	2021-12-31	-1.17	USD	
7155	ZYME	Zymeworks Inc	2022-05-03	2022-03-31	NaN	USD	
7156	ZYNE	Zynerba Pharmaceuticals Inc	2022-03-08	2021-12-31	-0.26	USD	
7157	ZYXI	Zynex Inc	2022-02-23	2021-12-31	0.20	USD	
7158	ZYXI	Zynex Inc	2022-04-27	2022-03-31	NaN	USD	

Figure 1.13: Preview of a DataFrame containing the downloaded earnings information

While getting the response to our API request is very similar to the previous example, handling the output is much different.

The output of `r.content` is a bytes object containing the output of the query as text. To mimic a normal file in-memory, we can use the `BytesIO` class from the `io` module. Then, we can normally load that mimicked file using the `pd.read_csv` function.



In the accompanying notebook, we present a few more functionalities of Alpha Vantage, such as getting the quarterly earnings data, downloading the calendar of the upcoming IPOs, and using `alpha_vantage`'s `TimeSeries` module to download stock price data.

See also

- <https://www.alphavantage.co/>—Alpha Vantage homepage
- <https://www.alphavantage.co/documentation/>—the API documentation
- https://github.com/RomelTorres/alpha_vantage—the GitHub repo of the third-party library used for accessing data from Alpha Vantage

Getting data from CoinGecko

The last data source we will cover is dedicated purely to cryptocurrencies. CoinGecko is a popular data vendor and crypto-tracking website, on which you can find real-time exchange rates, historical data, information about exchanges, upcoming events, trading volumes, and much more.

We can list a few of the advantages of CoinGecko:

- Completely free, and no need to register for an API key
- Aside from prices, it also provides updates and news about crypto
- It covers many coins, not only the most popular ones

In this recipe, we download Bitcoin's OHLC from the last 14 days.

How to do it...

Execute the following steps to download data from CoinGecko:

1. Import the libraries:

```
from pycoingecko import CoinGeckoAPI
from datetime import datetime
import pandas as pd
```

2. Instantiate the CoinGecko API:

```
cg = CoinGeckoAPI()
```

3. Get Bitcoin's OHLC prices from the last 14 days:

```
ohlc = cg.get_coin_ohlc_by_id(
    id="bitcoin", vs_currency="usd", days="14"
)
ohlc_df = pd.DataFrame(ohlc)
ohlc_df.columns = ["date", "open", "high", "low", "close"]
ohlc_df["date"] = pd.to_datetime(ohlc_df["date"], unit="ms")
ohlc_df
```

Running the snippet above returns the following DataFrame:

	date	open	high	low	close
0	2022-01-22 12:00:00	35631.29	35631.29	35631.29	35631.29
1	2022-01-22 16:00:00	35423.73	35952.33	35193.74	35193.74
2	2022-01-22 20:00:00	34991.02	35109.66	34527.65	34527.65
3	2022-01-23 00:00:00	34602.79	35630.21	34602.79	34935.31
4	2022-01-23 04:00:00	35180.44	35448.61	35044.59	35044.59
...
80	2022-02-04 20:00:00	39570.01	40557.17	39570.01	40495.25
81	2022-02-05 00:00:00	40781.16	40781.16	40573.23	40717.53
82	2022-02-05 04:00:00	41673.84	41673.84	41315.19	41454.15
83	2022-02-05 08:00:00	41492.85	41617.27	41492.85	41589.31
84	2022-02-05 12:00:00	41450.77	41554.04	41450.77	41554.04

Figure 1.14: Preview of the DataFrame containing the requested Bitcoin prices

In the preceding table, we can see that we have obtained the requested 14 days of data, sampled every 4 hours.

How it works...

After importing the libraries, we instantiated the CoinGeckoAPI object. Then, using its `get_coin_ohlc_by_id` method we downloaded the last 14 days' worth of BTC/USD exchange rates. It is worth mentioning there are some limitations of the API:

- We can only download data for a predefined number of days. We can select one of the following options: 1/7/14/30/90/180/365/max.
- The OHLC candles are sampled with a varying frequency depending on the requested horizon. They are sampled every 30 minutes for requests of 1 or 2 days. Between 3 and 30 days they are sampled every 4 hours. Above 30 days, they are sampled every 4 days.

The output of the `get_coin_ohlc_by_id` is a list of lists, which we can convert into a pandas DataFrame. We had to manually create the column names, as they were not provided by the API.

There's more...

We have seen that getting the OHLC prices can be a bit more difficult using the CoinGecko API as compared to the other vendors. However, CoinGecko has additional interesting information we can download using its API. In this section, we show a few possibilities.

Get the top 7 trending coins

We can use CoinGecko to acquire the top 7 trending coins—the ranking is based on the number of searches on CoinGecko within the last 24 hours. While downloading this information, we also get the coins' symbols, their market capitalization ranking, and the latest price in BTC:

```
trending_coins = cg.get_search_trending()
(
    pd.DataFrame([coin["item"] for coin in trending_coins["coins"]])
    .drop(columns=["thumb", "small", "large"])
)
```

Using the snippet above, we obtain the following DataFrame:

	id	coin_id	name	symbol	market_cap_rank	slug	price_btc	score
0	dogecoin	5	Dogecoin	DOGE	10	dogecoin	0.000004	0
1	civilization	17626	Civilization	CIV	674	civilization	0.000003	1
2	apecoin	24383	ApeCoin	APE	31	apecoin	0.000443	2
3	oasis-network	13162	Oasis Network	ROSE	110	oasis-network	0.000006	3
4	stepn	23597	STEPN	GMT	61	stepn	0.000081	4
5	unicrypt-2	12871	UniCrypt	UNCX	927	unicrypt	0.011535	5
6	xcad-network	15857	XCAD Network	XCAD	399	xcad-network	0.000107	6

Figure 1.15: Preview of the DataFrame containing the 7 trending coins and some information about them

Get Bitcoin's current price in USD

We can also extract current crypto prices in various currencies:

```
cg.get_price(ids="bitcoin", vs_currencies="usd")
```

Running the snippet above returns Bitcoin's real-time price:

```
{'bitcoin': {'usd': 47312}}
```

In the accompanying notebook, we present a few more functionalities of pycoingecko, such as getting the crypto price in different currencies than USD, downloading the entire list of coins supported on CoinGecko (over 9,000 coins), getting each coin's detailed market data (market capitalization, 24h volume, the all-time high, and so on), and loading the list of the most popular exchanges.

See also

You can find the documentation of the pycoingecko library here: <https://github.com/man-c/pycoingecko>.

Summary

In this chapter, we have covered a few of the most popular sources of financial data. However, this is just the tip of the iceberg. Below, you can find a list of other interesting data sources that might suit your needs even better.

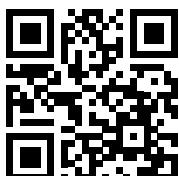
Additional data sources are:

- IEX Cloud (<https://iexcloud.io/>)—a platform providing a vast trove of different financial data. A notable feature that is unique to the platform is a daily and minutely sentiment score based on the activity on Stocktwits—an online community for investors and traders. However, that API is only available in the paid plan. You can access the IEX Cloud data using `pyex`, the official Python library.
- Tiingo (<https://www.tiingo.com/>) and the `tiingo` library.
- CryptoCompare (<https://www.cryptocompare.com/>)—the platform offers a wide range of crypto-related data via their API. What stands out about this data vendor is that they provide order book data.
- Twelve Data (<https://twelvedata.com/>).
- polygon.io (<https://polygon.io/>)—a trusted data vendor for real-time and historical data (stocks, forex, and crypto). Trusted by companies such as Google, Robinhood, and Revolut.
- Shrimpy (<https://www.shrimpy.io/>) and `shrimpy-python`—the official Python library for the Shrimpy Developer API.

In the next chapter, we will learn how to preprocess the downloaded data for further analysis.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

2

Data Preprocessing

You often hear in the data science industry that a data scientist typically spends close to 80% of their time on getting the data, processing it, cleaning it, and so on. And only then the remaining 20% of the time is actually spent on modeling, which is often considered to be the most interesting part. In the previous chapter, we have already learned how to download data from various sources. We still need to go through a few steps before we can draw actual insights from the data.

In this chapter, we will cover data preprocessing, that is, general wrangling/manipulation applied to the data before using it. The goal is not only to enhance the model's performance but also to ensure the validity of any analysis based on that data. In this chapter, we will focus on the financial time series, while in the subsequent chapters, we will also show how to work with other kinds of data.

In this chapter, we cover the following recipes:

- Converting prices to returns
- Adjusting the returns for inflation
- Changing the frequency of time series data
- Different ways of imputing missing data
- Changing currencies
- Different ways of aggregating trade data

Converting prices to returns

Many of the models and approaches used for time series modeling require the time series to be stationary. We will cover that topic in depth in *Chapter 6, Time Series Analysis and Forecasting*, however, we can get a quick glimpse of it now.

Stationarity assumes that the statistics (mathematical moments) of a process, such as the series' mean and variance, do not change over time. Using that assumption, we can build models that aim to forecast the future values of the process.

However, asset prices are usually non-stationary. Their statistics not only change over time, but we can also observe some trends (general patterns over time) or seasonality (patterns repeating over fixed time intervals). By transforming the prices into returns, we attempt to make the time series stationary.

Another benefit of using returns, as opposed to prices, is normalization. It means that we can easily compare various return series, which would not be that simple with raw stock prices, as one stock might start selling at \$10, while another at \$1,000.

There are two types of returns:

- Simple returns: They aggregate over assets—the simple return of a portfolio is the weighted sum of the returns of the individual assets in the portfolio. Simple returns are defined as:

$$R_t = (P_t - P_{t-1})/P_{t-1} = P_t/P_{t-1} - 1$$

- Log returns: They aggregate over time. It is easier to understand with the help of an example—the log return for a given month is the sum of the log returns of the days within that month. Log returns are defined as:

$$r_t = \log(P_t/P_{t-1}) = \log(P_t) - \log(P_{t-1})$$

P_t is the price of an asset in time t . In the preceding case, we do not consider dividends, which obviously impact the returns and require a small modification of the formulas.



The best practice while working with stock prices is to use adjusted values as they account for possible corporate actions, such as stock splits.

In general, log returns are often preferred over simple returns. Probably the most important reason for that is the fact that if we assume that the stock prices are log-normally distributed (which might or might not be the case for the particular time series), then the log returns would be normally distributed. And the normal distribution would work well with quite a lot of classic statistical approaches to time series modeling. Also, the difference between simple and log returns for daily/intraday data will be very small, in accordance with the general rule that log returns are smaller in value than simple returns.

In this recipe, we show how to calculate both types of returns using Apple's stock prices.

How to do it...

Execute the following steps to download Apple's stock prices and calculate simple/log returns:

1. Import the libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
```

2. Download the data and keep the adjusted close prices only:

```
df = yf.download("AAPL",
                  start="2010-01-01",
                  end="2020-12-31",
                  progress=False)
df = df.loc[:, ["Adj Close"]]
```

3. Calculate the simple and log returns using the adjusted close prices:

```
df["simple rtn"] = df["Adj Close"].pct_change()
df["log rtn"] = np.log(df["Adj Close"]/df["Adj Close"].shift(1))
```

4. Inspect the output:

```
df.head()
```

The resulting DataFrame looks as follows:

	Adj Close	simple rtn	log rtn
Date			
2009-12-31	6.444379	NaN	NaN
2010-01-04	6.544686	0.015565	0.015445
2010-01-05	6.556003	0.001729	0.001728
2010-01-06	6.451722	-0.015906	-0.016034
2010-01-07	6.439794	-0.001849	-0.001851

Figure 2.1: Snippet of the DataFrame containing Apple's adjusted close prices and simple/ log returns

The first row will always contain a **NaN (not a number)** value, as there is no previous price to use for calculating the returns.

How it works...

In *Step 2*, we downloaded price data from Yahoo Finance and only kept the adjusted close price for the calculation of the returns.

To calculate the simple returns, we used the `pct_change` method of pandas Series/DataFrame. It calculates the percentage change between the current and prior element (we can specify the number of lags, but for this specific case the default value of 1 suffices). Please bear in mind that the prior element is defined as the one in the row above the given row. In the case of working with time series data, we need to make sure that the data is sorted by the time index.

To calculate the log returns, we followed the formula given in the introduction to this recipe. When dividing each element of the series by its lagged value, we used the `shift` method with a value of 1 to access the prior element. In the end, we took the natural logarithm of the divided values by using the `np.log` function.

Adjusting the returns for inflation

When doing different kinds of analyses, especially long-term ones, we might want to consider inflation. **Inflation** is the general rise of the price level of an economy over time. Or to phrase it differently, the reduction of the purchasing power of money. That is why we might want to decouple the inflation from the increase of the stock prices caused by, for example, the companies' growth or development.

We can naturally adjust the prices of stocks directly, but in this recipe, we will focus on adjusting the returns and calculating the real returns. We can do so using the following formula:

$$R_t^r = \frac{1 + R_t}{1 + \pi_t} - 1$$

where R_t^r is the real return, R_t is the time t simple return, and π_t stands for the inflation rate.

For this example, we use Apple's stock prices from the years 2010 to 2020 (downloaded as in the previous recipe).

How to do it...

Execute the following steps to adjust the returns for inflation:

1. Import libraries and authenticate:

```
import pandas as pd
import nasdaqdatalink

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

2. Resample daily prices to monthly:

```
df = df.resample("M").last()
```

3. Download inflation data from Nasdaq Data Link:

```
df_cpi = (
    nasdaqdatalink.get(dataset="RATEINF/CPI_USA",
                        start_date="2009-12-01",
                        end_date="2020-12-31")
    .rename(columns={"Value": "cpi"})
)

df_cpi
```

Running the code generates the following table:

cpi	
Date	
2009-12-31	215.949
2010-01-31	216.687
2010-02-28	216.741
2010-03-31	217.631
2010-04-30	218.009
...	...
2020-08-31	259.918
2020-09-30	260.280
2020-10-31	260.388
2020-11-30	260.229
2020-12-31	260.474

Figure 2.2: Snippet of the DataFrame containing the values of the Consumer Price Index (CPI)

- Join inflation data to prices:

```
df = df.join(df_cpi, how="left")
```

- Calculate simple returns and inflation rate:

```
df["simple rtn"] = df["Adj Close"].pct_change()
df["inflation_rate"] = df["cpi"].pct_change()
```

- Adjust the returns for inflation and calculate the real returns:

```
df["real rtn"] = (
    (df["simple rtn"] + 1) / (df["inflation_rate"] + 1) - 1
)
df.head()
```

Running the code generates the following table:

	Adj Close	cpi	simple rtn	inflation_rate	real rtn
Date					
2009-12-31	6.444379	215.949	NaN	NaN	NaN
2010-01-31	5.873430	216.687	-0.088597	0.003417	-0.091701
2010-02-28	6.257530	216.741	0.065396	0.000249	0.065131
2010-03-31	7.186588	217.631	0.148470	0.004106	0.143774
2010-04-30	7.984450	218.009	0.111021	0.001737	0.109095

Figure 2.3: Snippet of the DataFrame containing the calculated inflation-adjusted returns

How it works...

First, we imported the libraries and authenticated with Nasdaq Data Link, which we used for downloading the inflation-related data. Then, we had to resample Apple's stock prices to a monthly frequency, as the inflation data is provided monthly. To do so, we chained the `resample` method with the `last` method. This way, we took the last price of the given month.

In *Step 3*, we downloaded the monthly **Consumer Price Index (CPI)** values from Nasdaq Data Link. It is a metric that examines the weighted average of prices of a basket of consumer goods and services, such as food, transportation, and so on.

Then, we used a left join to merge the two datasets (prices and CPI). A **left join** is a type of operation used for merging tables that returns all rows from the left table and the matched rows from the right table while leaving the unmatched rows empty.

By default, the `join` method uses the indices of the tables to carry out the actual joining. We can use the `on` argument to specify which column/columns to use otherwise.

Having all the data in one DataFrame, we used the `pct_change` method to calculate the simple returns and the inflation rate. Lastly, we used the formula presented in the introduction to calculate the real returns.

There's more...

We have already explored how to download the inflation data from Nasdaq Data Link. Alternatively, we can use a handy library called `cpi`.

1. Import the library:

```
import cpi
```

At this point, we might encounter the following warning:

```
StaleDataWarning: CPI data is out of date
```

If that is the case, we just need to run the following line of code to update the data:

```
cpi.update()
```

2. Obtain the default CPI series:

```
cpi_series = cpi.series.get()
```

Here we download the default CPI index (`CUUR0000SA0`: All items in U.S. city average, all urban consumers, not seasonally adjusted), which will work for most of the cases. Alternatively, we can provide the `items` and `area` arguments to download a more tailor-made series. We can also use the `get_by_id` function to download a particular CPI series.

3. Convert the object into a pandas DataFrame:

```
df_cpi_2 = cpi_series.to_dataframe()
```

4. Filter the DataFrame and view the top 12 observations:

```
df_cpi_2.query("period_type == 'monthly' and year >= 2010") \
    .loc[:, ["date", "value"]] \
    .set_index("date") \
    .head(12)
```

Running the code generates the following output:

	value
date	
2010-01-01	216.687
2010-02-01	216.741
2010-03-01	217.631
2010-04-01	218.009
2010-05-01	218.178
2010-06-01	217.965
2010-07-01	218.011
2010-08-01	218.312
2010-09-01	218.439
2010-10-01	218.711
2010-11-01	218.803
2010-12-01	219.179

Figure 2.4: The first 12 values of the DataFrame containing the downloaded values of the CPI

In this step, we used some filtering to compare the data to the data downloaded before from Nasdaq Data Link. We used the `query` method to only keep the monthly data from the year 2010 onward. We displayed only two selected columns and the first 12 observations, for comparison's sake.

We will also be using the `cpi` library in later chapters to directly inflate the prices using the `inflate` function.

See also

- <https://github.com/palewire/cpi>—the GitHub repo of the `cpi` library

Changing the frequency of time series data

When working with time series, and especially financial ones, we often need to change the frequency (periodicity) of the data. For example, we receive daily OHLC prices, but our algorithm works with weekly data. Or we have daily alternative data, and we want to match it with our live feed of intraday data.

The general rule of thumb for changing frequency can be broken down into the following:

- Multiply/divide the log returns by the number of time periods.
- Multiply/divide the volatility by the square root of the number of time periods.



For any process with independent increments (for example, the geometric Brownian motion), the variance of the logarithmic returns is proportional to time. For example, the variance of $r_{t3} - r_{t1}$ is going to be the sum of the following two variances: $r_{t2} - r_{t1}$ and $r_{t3} - r_{t2}$, assuming $t_1 \leq t_2 \leq t_3$. In such a case, when we also assume that the parameters of the process do not change over time (homogeneity) we arrive at the proportionality of the variance to the length of the time interval. Which in practice means that the standard deviation (volatility) is proportional to the square root of time.

In this recipe, we present an example of how to calculate the monthly realized volatilities for Apple using daily returns and then annualize the values. We can often encounter annualized volatility when looking at the risk-adjusted performance of an investment.

The formula for realized volatility is as follows:

$$RV = \sqrt{\sum_{i=1}^T r_t^2}$$

Realized volatility is frequently used for calculating the daily volatility using intraday returns.

The steps we need to take are as follows:

- Download the data and calculate the log returns
- Calculate the realized volatility over the months
- Annualize the values by multiplying by $\sqrt{12}$, as we are converting from monthly values

Getting ready

We assume you have followed the instructions from the previous recipes and have a DataFrame called `df` with a single `log rtn` column and timestamps as the index.

How to do it...

Execute the following steps to calculate and annualize the monthly realized volatility:

1. Import the libraries:

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

2. Define the function for calculating the realized volatility:

```
def realized_volatility(x):
    return np.sqrt(np.sum(x**2))
```

3. Calculate the monthly realized volatility:

```
df_rv = (  
    df.groupby(pd.Grouper(freq="M"))  
    .apply(realized_volatility)  
    .rename(columns={"log rtn": "rv"})  
)
```

4. Annualize the values:

```
df_rv.rv = df_rv["rv"] * np.sqrt(12)
```

5. Plot the results:

```
fig, ax = plt.subplots(2, 1, sharex=True)  
ax[0].plot(df)  
ax[0].set_title("Apple's log returns (2000-2012)")  
ax[1].plot(df_rv)  
ax[1].set_title("Annualized realized volatility")  
  
plt.show()
```

Executing the snippet results in the following plots:

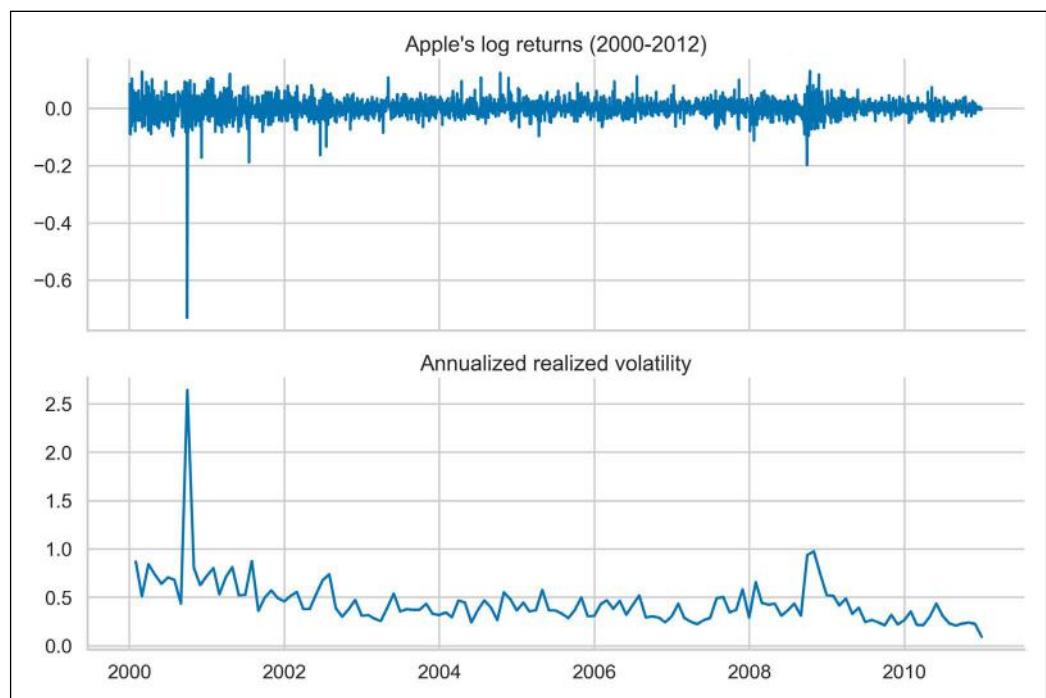


Figure 2.5: Apple's log return series and the corresponding realized volatility (annualized)

We can see that the spikes in the realized volatility coincide with some extreme returns (which might be outliers).

How it works...

Normally, we could use the `resample` method of a pandas DataFrame. Supposing we wanted to calculate the average monthly return, we could use `df["log_rtn"].resample("M").mean()`.

With the `resample` method, we can use any built-in aggregate function of pandas, such as `mean`, `sum`, `min`, and `max`. However, our case at hand is a bit more complex so we first defined a helper function called `realized_volatility`. Because we wanted to use a custom function for aggregation, we replicated the behavior of `resample` by using a combination of `groupby`, `Grouper`, and `apply`.

We presented the most basic visualization of the results (please refer to *Chapter 3, Visualizing Financial Time Series*, for information about visualizing time series).

Different ways of imputing missing data

While working with any time series, it can happen that some data is missing, due to many possible reasons (someone forgot to input the data, a random issue with the database, and so on). One of the available solutions would be to discard observations with missing values. However, imagine a scenario in which we are analyzing multiple time series at once, and only one of the series is missing a value due to some random mistake. Do we still want to remove all the other potentially valuable pieces of information because of this single missing value? Probably not. And there are many other potential scenarios in which we would rather treat the missing values somehow, rather than discarding those observations.

Two of the simplest approaches to imputing missing time series data are:

- Backward filling—fill the missing value with the next known value
- Forward filling—fill the missing value with the previous known value

In this recipe, we show how to use those techniques to easily deal with missing values in the example of the CPI time series.

How to do it...

Execute the following steps to try out different ways of imputing missing data:

1. Import the libraries:

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

2. Download the inflation data from Nasdaq Data Link:

```
nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
df = (
    nasdaqdatalink.get(dataset="RATEINF/CPI_USA",
                        start_date="2015-01-01",
                        end_date="2020-12-31")
    .rename(columns={"Value": "cpi"})
)
```

3. Introduce five missing values at random:

```
np.random.seed(42)
rand_indices = np.random.choice(df.index, 5, replace=False)

df["cpi_missing"] = df.loc[:, "cpi"]
df.loc[rand_indices, "cpi_missing"] = np.nan
df.head()
```

In the following table, we can see we have successfully introduced missing values into the data:

	cpi	cpi_missing
Date		
2015-01-31	233.707	NaN
2015-02-28	234.722	234.722
2015-03-31	236.119	236.119
2015-04-30	236.599	236.599
2015-05-31	237.805	NaN

Figure 2.6: Preview of the DataFrame with downloaded CPI data and the added missing values

4. Fill in the missing values using different methods:

```
for method in ["bfill", "ffill"]:
    df[f"method_{method}"] = (
        df[["cpi_missing"]].fillna(method=method)
    )
```

5. Inspect the results by displaying the rows in which we created the missing values:

```
df.loc[rand_indices].sort_index()
```

Running the code results in the following output:

Date	cpi	cpi_missing	method_bfill	method_ffill
2015-01-31	233.707	NaN	234.722	NaN
2015-05-31	237.805	NaN	238.638	236.599
2016-07-31	240.647	NaN	240.849	241.038
2017-05-31	244.733	NaN	244.955	244.524
2020-03-31	258.115	NaN	256.389	258.678

Figure 2.7: Preview of the DataFrame after imputing the missing values

We can see that backward filling worked for all the missing values we created. However, forward filling failed to impute one value. That is because this is the first data point in the series, so there is no available value to fill forward.

- Plot the results for the years 2015 to 2016:

```
df.loc[:"2017-01-01"] \
    .drop(columns=["cpi_missing"]) \
    .plot(title="Different ways of filling missing values");
```

Running the snippet generates the following plot:

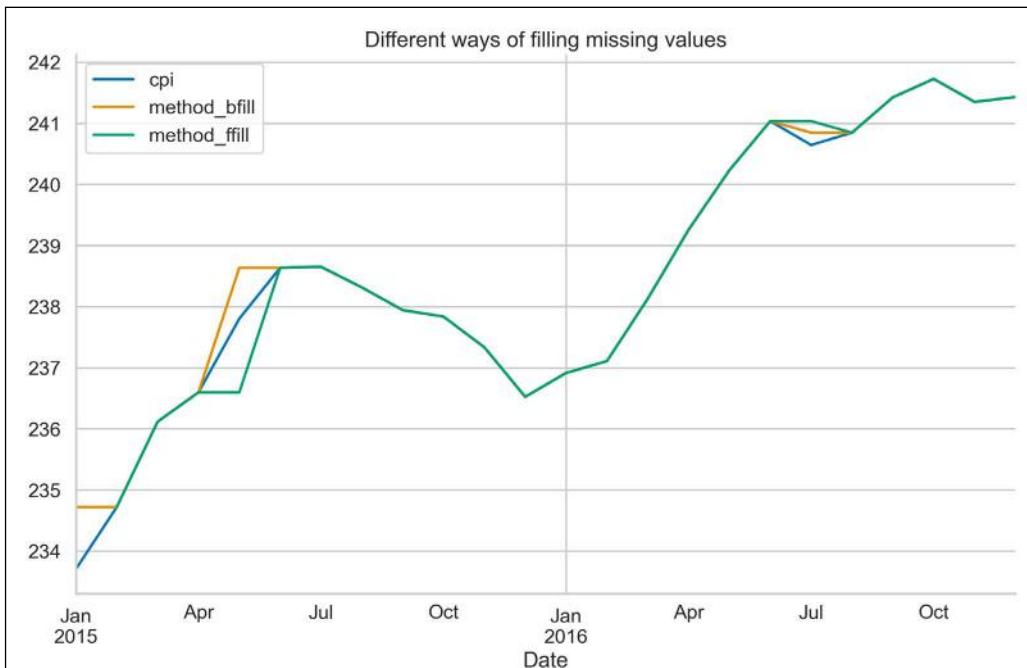


Figure 2.8: The comparison of backward and forward filling on the CPI time series

In *Figure 2.8*, we can clearly see how both forward and backward filling work in practice.

How it works...

After importing the libraries, we downloaded the 6 years of monthly CPI data from Nasdaq Data Link. Then, we selected 5 random indices from the DataFrame to artificially create missing values. To do so, we replaced those values with NaNs.

In *Step 4*, we applied two different imputation methods to our time series. We used the `fillna` method of a pandas DataFrame and specified the `method` argument as `bfill` (backward filling) or `ffill` (forward filling). We saved the imputed series as new columns, in order to clearly compare the results. Please remember that the `fillna` method replaces the missing values and keeps the other values intact.

Instead of providing a method of filling the missing data, we could have specified a value of our choice, for example, `0` or `999`. However, using an arbitrary number might not make much sense in the case of time series data, so that is not advised.



We used `np.random.seed(42)` to make the experiment reproducible. Each time you run this cell, you will get the same random numbers. You can use any number for the seed and the random choice will be different for each of those.

In *Step 5*, we inspected the imputed values. For brevity, we have only displayed the indices we have randomly selected. We used the `sort_index` method to sort them by the date. This way, we can clearly see that the first value was not filled using the forward filling, as it is the very first observation in the time series.

Lastly, we plotted all the time series from the years 2015 to 2016. In the plot, we can clearly see how backward/forward filling imputes the missing values.

There's more...

In this recipe, we have explored some simple methods of imputing missing data. Another possibility is to use interpolation, to which there are many different approaches. As such, in this example, we will use the linear one. Please refer to the pandas documentation (the link is available in the *See also* subsection) for more information about the available methods of interpolation.

1. Use linear interpolation to fill the missing values:

```
df[ "method_interpolate" ] = df[ [ "cpi_missing" ] ].interpolate()
```

2. Inspect the results:

```
df.loc[ rand_indices ].sort_index()
```

Running the snippet generates the following output:

Date	cpi	cpi_missing	method_bfill	method_ffill	method_interpolate
2015-01-31	233.707	NaN	234.722	NaN	NaN
2015-05-31	237.805	NaN	238.638	236.599	237.6185
2016-07-31	240.647	NaN	240.849	241.038	240.9435
2017-05-31	244.733	NaN	244.955	244.524	244.7395
2020-03-31	258.115	NaN	256.389	258.678	257.5335

Figure 2.9: Preview of the DataFrame after imputing the missing values with linear interpolation

Unfortunately, linear interpolation also cannot deal with the missing value located at the very beginning of the time series.

3. Plot the results:

```
df.loc[:"2017-01-01"] \
    .drop(columns=["cpi_missing"]) \
    .plot(title="Different ways of filling missing values");
```

Running the snippet generates the following plot:



Figure 2.10: The comparison of backward and forward filling on the CPI time series, including interpolation

In *Figure 2.10*, we can see how linear interpolation connects the known observations with a straight line to impute the missing value.

In this recipe, we explored imputing missing data for time series data. However, these are not all of the possible approaches. We could have, for example, used the moving average of the last few observations to impute any missing values. There are certainly a lot of possible methodologies to choose from. In *Chapter 13, Applied Machine Learning: Identifying Credit Default*, we will show how to approach the issue of missing values for other kinds of datasets.

See also

- <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html>—here you can see all the available methods of interpolating available in pandas.

Converting currencies

Another quite common preprocessing step you might encounter while working on financial tasks is converting currencies. Imagine you have a portfolio of multiple assets, priced in different currencies and you would like to arrive at a total portfolio's worth. The simplest example might be American and European stocks.

In this recipe, we show how to easily convert stock prices from USD to EUR. However, the very same steps can be used to convert any pair of currencies.

How to do it...

Execute the following steps to convert stock prices from USD to EUR:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
from forex_python.converter import CurrencyRates
```

2. Download Apple's OHLC prices from January 2020:

```
df = yf.download("AAPL",
                 start="2020-01-01",
                 end="2020-01-31",
                 progress=False)
df = df.drop(columns=["Adj Close", "Volume"])
```

3. Instantiate the `CurrencyRates` object:

```
c = CurrencyRates()
```

4. Download the USD/EUR rate for each required date:

```
df["usd_eur"] = [c.get_rate("USD", "EUR", date) for date in df.index]
```

5. Convert the prices in USD to EUR:

```
for column in df.columns[:-1]:
    df[f'{column}_EUR'] = df[column] * df['usd_eur']
df.head()
```

Running the snippet generates the following preview:

Date	Open	High	Low	Close	usd_eur	Open_EUR	High_EUR	Low_EUR	Close_EUR
2019-12-31	72.482	73.420	72.380	73.412	0.890	64.521	65.355	64.429	65.348
2020-01-02	74.060	75.150	73.798	75.088	0.893	66.166	67.140	65.932	67.084
2020-01-03	74.287	75.145	74.125	74.357	0.897	66.643	67.413	66.498	66.706
2020-01-06	73.448	74.990	73.188	74.950	0.893	65.613	66.991	65.381	66.956
2020-01-07	74.960	75.225	74.370	74.598	0.895	67.096	67.334	66.568	66.772

Figure 2.11: Preview of the DataFrame containing the original prices in USD and the ones converted to EUR

We can see that we have successfully converted all four columns with prices into EUR.

How it works...

In *Step 1*, we have imported the required libraries. Then, we downloaded Apple's OHLC prices from January 2020 using the already covered `yfinance` library.

In *Step 3*, we instantiated the `CurrencyRates` object from the `forex-python` library. Under the hood, the library is using the Forex API (<https://theforexapi.com>), which is a free API for accessing current and historical foreign exchange rates published by the European Central Bank.

In *Step 4*, we used the `get_rate` method to download the USD/EUR exchange rates for all the dates available in the DataFrame with stock prices. To do so efficiently, we used list comprehension and stored the outputs in a new column. One potential drawback of the library and the present implementation is that we need to download each and every exchange rate individually, which might not be scalable for large DataFrames.



While using the library, you can sometimes run into the following error: `RatesNotFoundError: Currency Rates Source Not Ready`. The most probable cause is that you are trying to get the exchange rates from weekends. The easiest solution is to skip those days in the list comprehension/for loop and fill in the missing values using one of the approaches covered in the previous recipe.

In the last step, we iterated over the columns of the initial DataFrame (all except the exchange rate) and multiplied the USD price by the exchange rate. We stored the outcomes in new columns, with `_EUR` subscript.

There's more...

Using the `forex_python` library, we can easily download the exchange rates for many currencies at once. To do so, we can use the `get_rates` method. In the following snippet, we download the current exchange rates of USD to the 31 available currencies. We can naturally specify the date of interest, just as we have done before.

1. Get the current USD exchange rates to 31 available currencies:

```
usd_rates = c.get_rates("USD")
usd_rates
```

The first five entries look as follows:

```
{'EUR': 0.8441668073611345,
'JPY': 110.00337666722943,
'BGN': 1.651021441836907,
'CZK': 21.426641904440316,
'DKK': 6.277224379537396,
}
```

In this recipe, we have mostly focused on the `forex_python` library, as it is quite handy and flexible. However, we might download historical exchange rates from many different sources and arrive at the same results (accounting for some margin of error depending on the data provider). Quite a few of the data providers described in *Chapter 1, Acquiring Financial Data*, provide historical exchange rates. Below, we show how to get those rates using Yahoo Finance.

2. Download the USD/EUR exchange rate from Yahoo Finance:

```
df = yf.download("USDEUR=X",
                  start="2000-01-01",
                  end="2010-12-31",
                  progress=False)
df.head()
```

Running the snippet results in the following output:

Date	Open	High	Low	Close	Adj Close	Volume
2003-12-01	0.83098	0.83724	0.83056	0.83577	0.83577	0
2003-12-02	0.83605	0.83710	0.82583	0.82720	0.82720	0
2003-12-03	0.82713	0.82802	0.82440	0.82488	0.82488	0
2003-12-04	0.82508	0.83029	0.82345	0.82775	0.82775	0
2003-12-05	0.82795	0.82878	0.82028	0.82055	0.82055	0

Figure 2.12: Preview of the DataFrame with the downloaded exchange rates

In *Figure 2.12*, we can see one of the limitations of this data source—the data for this currency pair is only available since December 2003. Also, Yahoo Finance is providing the OHLC variant of the exchange rates. To arrive at a single number used for conversion, you can pick any of the four values (depending on the use case) or calculate the mid-value (the middle between low and high values).

See also

- <https://github.com/MicroPyramid/forex-python>—the GitHub repo of the `forex-python` library

Different ways of aggregating trade data

Before diving into building a machine learning model or designing a trading strategy, we not only need reliable data, but we also need to aggregate it into a format that is convenient for further analysis and appropriate for the models we choose. The term **bars** refers to a data representation that contains basic information about the price movements of any financial asset. We have already seen one form of bars in *Chapter 1, Acquiring Financial Data*, in which we explored how to download financial data from a variety of sources.

There, we downloaded OHLCV data sampled by some time period, be it a month, day, or intraday frequencies. This is the most common way of aggregating financial time series data and is known as the **time bars**.

There are some drawbacks of sampling financial time series by time:

- Time bars disguise the actual rate of activity in the market—they tend to oversample low activity periods (for example, noon) and undersample high activity periods (for example, close to market open and close).
- Nowadays, markets are more and more controlled by trading algorithms and bots, so they no longer follow human daylight cycles.
- Time-based bars offer poorer statistical properties (for example, serial correlation, heteroskedasticity, and non-normality of returns).
- Given that this is the most popular kind of aggregation and the easiest one to access, it can also be prone to manipulation (for example, iceberg orders).



Iceberg orders are large orders that were divided into smaller limit orders to hide the actual order quantity. They are called “iceberg orders” because the visible orders are just the “tip of the iceberg,” while a significant number of limit orders is waiting, ready to be placed.

To overcome those issues and gain a competitive edge, practitioners also use other kinds of aggregation. Ideally, they would want to have a bar representation in which each bar contains the same amount of information. Some of the alternatives they are using include:

- **Tick bars**—named after the fact that transactions/trades in financial markets are often referred to as ticks. For this kind of aggregation, we sample an OHLCV bar every time a predefined number of transactions occurs.
- **Volume bars**—we sample a bar every time a predefined volume (measured in any unit, for example, shares, coins, etc.) is exchanged.
- **Dollar bars**—we sample a bar every time a predefined dollar amount is exchanged. Naturally, we can use any other currency of choice.

Each of these forms of aggregations has its strengths and weaknesses that we should be aware of.

Tick bars offer a better way of tracking the actual activity in the market, together with the volatility. However, a potential issue arises out of the fact that one trade can contain any number of units of a certain asset. So, a buy order of a single share is treated equally to an order of 10,000 shares.

Volume bars are an attempt at overcoming this problem. However, they come with an issue of their own. They do not correctly reflect situations in which asset prices change significantly or when stock splits happen. This makes them unreliable for comparison between periods affected by such situations.

That is where the third type of bar comes into play—the **dollar bars**. It is often considered the most robust way of aggregating price data. Firstly, the dollar bars help bridge the gap with price volatility, which is especially important for highly volatile markets such as cryptocurrencies. Then, sampling by dollars is helpful to preserve the consistency of information. The second reason is that dollar bars are resistant to the outstanding amount of the security, so they are not affected by actions such as stock splits, corporate buybacks, issuance of new shares, and so on.

In this recipe, we will learn how to create all four types of bars mentioned above using trade data coming from Binance, one of the most popular cryptocurrency exchanges. We decided to use cryptocurrency data as it is much easier to obtain (free of charge) compared to, for example, equity data. However, the presented methodology remains the same for other asset classes as well.

How to do it...

Execute the following steps to download trade data from Binance and aggregate it into four different kinds of bars:

1. Import the libraries:

```
from binance.spot import Spot as Client
import pandas as pd
import numpy as np
```

2. Instantiate the Binance client and download the last 500 BTCEUR trades:

```
spot_client = Client(base_url="https://api3.binance.com")
r = spot_client.trades("BTCEUR")
```

3. Process the downloaded trades into a pandas DataFrame:

```
df = (
    pd.DataFrame(r)
    .drop(columns=["isBuyerMaker", "isBestMatch"])
)
df["time"] = pd.to_datetime(df["time"], unit="ms")

for column in ["price", "qty", "quoteQty"]:
    df[column] = pd.to_numeric(df[column])
df
```

Executing the code returns the following DataFrame:

	id	price	qty	quoteQty	time
0	77999355	33285.50	0.00288	95.862240	2022-02-23 19:41:15.896
1	77999356	33286.84	0.00336	111.843782	2022-02-23 19:41:15.896
2	77999357	33275.29	0.00813	270.528108	2022-02-23 19:41:15.941
3	77999358	33277.44	0.01001	333.107174	2022-02-23 19:41:17.896
4	77999359	33275.29	0.01001	333.085653	2022-02-23 19:41:17.900
...
495	77999850	33268.98	0.00067	22.290217	2022-02-23 19:50:56.806
496	77999851	33268.49	0.00249	82.838540	2022-02-23 19:50:56.806
497	77999852	33268.98	0.02126	707.298515	2022-02-23 19:50:58.903
498	77999853	33268.00	0.00596	198.277280	2022-02-23 19:50:58.903
499	77999854	33268.52	0.00150	49.902780	2022-02-23 19:50:59.291

Figure 2.13: The DataFrame containing the last 500 BTC-EUR transactions

We can see that 500 transactions in the BTCEUR market happened over a span of approximately nine minutes. For more popular markets, this window can be significantly reduced. The qty column contains the traded amount of BTC, while quoteQty contains the EUR price of the traded quantity, which is the same as multiplying the price column by the qty column.

4. Define a function aggregating the raw trades information into bars:

```
def get_bars(df, add_time=False):
    ohlc = df[["price"]].ohlc()
    vwap = (
        df.apply(lambda x: np.average(x["price"], weights=x["qty"]))
        .to_frame("vwap")
    )
    vol = df[["qty"]].sum().to_frame("vol")
    cnt = df[["qty"]].size().to_frame("cnt")
```

```

if add_time:
    time = df[time].last().to_frame("time")
    res = pd.concat([time, ohlc, vwap, vol, cnt], axis=1)
else:
    res = pd.concat([ohlc, vwap, vol, cnt], axis=1)
return res

```

5. Get the time bars:

```

df_grouped_time = df.groupby(pd.Grouper(key="time", freq="1Min"))
time_bars = get_bars(df_grouped_time)
time_bars

```

Running the code generates the following time bars:

	open	high	low	close	vwap	vol	cnt
time							
2022-02-23 19:41:00	33285.50	33286.84	33254.00	33269.47	33269.239073	0.71050	52
2022-02-23 19:42:00	33265.24	33265.63	33226.18	33231.76	33237.848521	2.26604	110
2022-02-23 19:43:00	33234.74	33250.30	33215.73	33240.09	33231.975184	0.70111	73
2022-02-23 19:44:00	33240.10	33240.10	33216.23	33231.33	33234.523449	0.81760	21
2022-02-23 19:45:00	33227.68	33286.04	33226.31	33279.94	33245.756842	3.39557	89
2022-02-23 19:46:00	33270.23	33305.45	33266.54	33279.62	33284.459090	0.46893	31
2022-02-23 19:47:00	33283.24	33328.46	33273.90	33322.05	33295.109170	0.30348	32
2022-02-23 19:48:00	33308.50	33333.98	33297.37	33315.46	33315.803656	0.58047	37
2022-02-23 19:49:00	33310.42	33322.00	33281.73	33294.93	33309.609233	0.22579	21
2022-02-23 19:50:00	33283.21	33294.20	33268.00	33268.52	33281.789160	0.73358	34

Figure 2.14: Preview of the DataFrame with time bars

6. Get the tick bars:

```

bar_size = 50
df["tick_group"] = (
    pd.Series(list(range(len(df))))
    .div(bar_size)
    .apply(np.floor)
    .astype(int)
    .values
)
df_grouped_ticks = df.groupby("tick_group")
tick_bars = get_bars(df_grouped_ticks, add_time=True)
tick_bars

```

Running the code generates the following tick bars:

tick_group		time	open	high	low	close	vwap	vol	cnt
0	2022-02-23 19:41:57.784	33285.50	33286.84	33254.00	33270.89	33269.220356	0.65723	50	
1	2022-02-23 19:42:29.168	33269.47	33269.47	33243.50	33243.50	33249.679200	0.75967	50	
2	2022-02-23 19:42:43.886	33241.86	33244.71	33226.18	33230.04	33233.334328	1.45604	50	
3	2022-02-23 19:43:37.837	33234.89	33244.70	33215.73	33222.93	33228.599024	0.31810	50	
4	2022-02-23 19:44:28.440	33222.72	33250.30	33216.23	33216.98	33234.604510	1.26310	50	
5	2022-02-23 19:45:15.153	33226.01	33258.29	33221.43	33233.13	33236.663489	1.87995	50	
6	2022-02-23 19:46:02.625	33233.13	33286.04	33233.13	33284.14	33256.233756	1.56017	50	
7	2022-02-23 19:47:47.964	33284.89	33305.45	33266.54	33293.35	33284.115017	0.63715	50	
8	2022-02-23 19:49:05.929	33294.13	33333.98	33294.13	33314.63	33314.426886	0.78777	50	
9	2022-02-23 19:50:59.291	33313.24	33322.00	33268.00	33268.52	33286.482830	0.88389	50	

Figure 2.15: Preview of the DataFrame with tick bars

We can see that each group contains exactly 50 trades, just as we intended.

7. Get the volume bars:

```
bar_size = 1
df["cum_qty"] = df["qty"].cumsum()
df["vol_group"] = (
    df["cum_qty"]
    .div(bar_size)
    .apply(np.floor)
    .astype(int)
    .values
)
df_grouped_ticks = df.groupby("vol_group")
volume_bars = get_bars(df_grouped_ticks, add_time=True)
volume_bars
```

Running the code generates the following volume bars:

vol_group		time	open	high	low	close	vwap	vol	cnt
0	2022-02-23 19:42:19.499	33285.50	33286.84	33246.90	33250.00	33264.436711	0.99446	85	
1	2022-02-23 19:42:31.215	33246.91	33253.07	33226.18	33226.87	33240.193454	0.86169	46	
2	2022-02-23 19:43:04.416	33232.23	33244.71	33230.04	33239.77	33232.363594	1.12313	33	
3	2022-02-23 19:44:01.139	33240.00	33250.30	33215.73	33240.10	33232.135781	0.71425	72	
4	2022-02-23 19:45:12.130	33240.10	33258.29	33216.23	33241.21	33238.411739	1.25593	44	
5	2022-02-23 19:45:15.146	33241.21	33241.21	33232.03	33233.13	33234.773416	0.83403	15	
6	2022-02-23 19:45:21.810	33233.13	33253.41	33233.13	33248.69	33236.027748	1.21260	20	
7	2022-02-23 19:46:23.762	33248.69	33286.04	33242.90	33281.10	33270.171474	0.99072	45	
8	2022-02-23 19:48:32.173	33280.33	33333.98	33273.90	33301.66	33300.233202	0.98864	69	
9	2022-02-23 19:50:36.613	33301.65	33322.00	33274.39	33283.79	33296.304224	0.94414	53	
10	2022-02-23 19:50:59.291	33283.81	33294.20	33268.00	33268.52	33280.873796	0.28348	18	

Figure 2.16: Preview of the DataFrame with volume bars

We can see that all the bars contain approximately the same volume. The last one is a bit smaller, simply because we did not have enough total volume in the 500 trades.

8. Get the dollar bars:

```
bar_size = 50000
df["cum_value"] = df["quoteQty"].cumsum()
df["value_group"] = (
    df["cum_value"]
    .div(bar_size)
    .apply(np.floor)
    .astype(int)
    .values
)
df_grouped_ticks = df.groupby("value_group")
dollar_bars = get_bars(df_grouped_ticks, add_time=True)
dollar_bars
```

Running the code generates the following dollar bars:

	time	open	high	low	close	vwap	vol	cnt
value_group								
0	2022-02-23 19:42:29.230	33285.50	33286.84	33239.96	33239.96	33258.572184	1.42997	103
1	2022-02-23 19:43:17.962	33239.96	33244.71	33226.18	33231.41	33233.221296	1.57424	66
2	2022-02-23 19:44:58.211	33231.40	33250.30	33215.73	33231.33	33233.232079	1.49104	87
3	2022-02-23 19:45:15.146	33227.68	33258.29	33226.31	33233.13	33238.544792	1.28824	39
4	2022-02-23 19:45:59.597	33233.13	33286.04	33233.13	33280.92	33243.191526	1.70740	43
5	2022-02-23 19:48:32.173	33279.94	33333.98	33266.54	33301.66	33293.331944	1.48456	91
6	2022-02-23 19:50:59.291	33301.65	33322.00	33268.00	33268.52	33292.741055	1.22762	71

Figure 2.17: Preview of the DataFrame with dollar bars

How it works...

After importing the libraries, we instantiated the Binance client and downloaded the 500 most recent trades in the BTCEUR market using the `trades` method of the Binance client. We chose this one on purpose, as it is not as popular as BTCUSD and the default 500 trades actually span a few minutes. We could increase the number of trades up to 1,000 using the `limit` argument.



We have used the easiest way to download the 500 most recent trades. However, we could do better and recreate the trades over a longer period of time. To do so, we could use the `historical_trades` method. It contains an additional argument called `fromId`, which we could use to specify from which particular trade we would like to start our batch download. Then, we could chain those API calls using the last known ID to recreate the trade history from a longer period of time. However, to do so, we need to have a Binance account, create personal API keys, and provide them to the `Client` class.

In *Step 3*, we prepared the data for further analysis, that is, we converted the response from the Binance client into a pandas DataFrame, dropped two columns we will not be using, converted the time column into datetime, and converted to columns containing prices and quantities into numeric ones, as they were expressed as object type, which is a string.

Then, we defined a helper function for calculating the bars per some group. The input of the function must be a DataFrameGroupBy object, that is, the output of applying the groupby method to a pandas DataFrame. That is because the function calculates a bunch of aggregate statistics:

- OHLC values using the ohlc method.
- The **volume-weighted average price (VWAP)** by applying the np.average method and using the quantity of the trade as the weights argument.
- The total volume as the sum of the traded quantity.
- The number of trades in a bar by using the size method.
- Optionally, the function also returns the timestamp of the bar, which is simply the last timestamp of the group.

All of those are separate DataFrames, which we ultimately concatenated using the pd.concat function.

In *Step 5*, we calculated the time bars. We had to use the groupby method combined with pd.Grouper. We indicated we want to create the groups on the time column and used a one-minute frequency. Then, we passed the DataFrameGroupBy object to our get_bars function, which returned the time bars.

In *Step 6*, we calculated the tick bars. The process was slightly different than with time bars, as we first had to create the column on which we want to group the trades. The idea was that we group the trades in blocks of 50 (this is an arbitrary number and should be determined according to the logic of the analysis). To create such groups, we divided the row number by the chosen bar size, rounded the result down (using np.floor), and converted it into an integer. Then, we grouped the trades using the newly created column and applied the get_bars function.

In *Step 7*, we calculated the volume bars. The process was quite similar to the tick bars. The difference was in creating the grouping column, which this time was based on the cumulative sum of the traded quantity. We selected the bar size of 1 BTC.

The last step was to calculate the dollar bars. The process was almost identical to the volume bars, but we created the grouping column by applying a cumulative sum to the quoteQty column, instead of the qty one used before.

There's more...

The list of alternative kinds of bars in this recipe is not exhaustive. For example, De Prado (2018) suggests using **imbalance bars**, which attempt to sample the data when there is an imbalance of buying/selling activity, as this might imply information asymmetry between market participants. The reasoning behind those bars is market participants either buy or sell large quantities of a given asset, but they do not frequently do both simultaneously. Hence, sampling when imbalance events occur helps to focus on large movements and pay less attention to periods without interesting activity.

See also

- De Prado, M. L. (2018). *Advances in Financial Machine Learning*. John Wiley & Sons.
- <https://github.com/binance/binance-connector-python>—the GitHub repo of the library used for connecting to Binance’s API

Summary

In this chapter, we have learned how to preprocess financial time series data. We started by showing how to calculate returns and potentially adjust them for inflation. Then, we covered a few of the popular methods for imputing missing values. Lastly, we explained the different approaches to aggregating trade data and why choosing the correct one matters.

We should always pay significant attention to this step, as we not only want to enhance our model’s performance but also to ensure the validity of any analysis. In the next chapter, we will continue working with the preprocessed data and learn how to create time series visualization.

3

Visualizing Financial Time Series

The old adage *a picture is worth a thousand words* is very much applicable in the data science field. We can use different kinds of plots to not only explore data but also tell data-based stories.

While working with financial time series data, quickly plotting the series can already lead to many valuable insights, such as:

- Is the series continuous?
- Are there any unexpected missing values?
- Do some values look like outliers?
- Are there any patterns we can quickly see and use for further analyses?

Naturally, these are only some of the potential questions that aim to help us with our analyses. The main goal of visualization at the very beginning of any project is to familiarize yourself with the data and get to know it a bit better. And only then can we move on to conducting proper statistical analysis and building machine learning models that aim to predict the future values of the series.

Regarding data visualization, Python offers a variety of libraries that can get the job done, with various levels of required complexity (including the learning curve) and slightly different quality of the outputs. Some of the most popular libraries used for visualization include:

- `matplotlib`
- `seaborn`
- `plotly`
- `altair`
- `plotnine`—This library is based on R's `ggplot`, so might be especially interesting for those who are also familiar with R
- `bokeh`

In this chapter, we will use quite a few of the libraries mentioned above. We believe that it makes sense to use the best tool for the job, so if it takes a one-liner to create a certain plot in one library while it takes 20 lines in another one, then the choice is quite clear. You can most likely create all the visualizations shown in this chapter using any of the mentioned libraries.



If you need to create a very custom plot that is not provided out-of-the-box in one of the most popular libraries, then `matplotlib` should be your choice, as you can create pretty much anything using it.

In this chapter, we will cover the following recipes:

- Basic visualization of time series data
- Visualizing seasonal patterns
- Creating interactive visualizations
- Creating a candlestick chart

Basic visualization of time series data

The most common starting point of visualizing time series data is a simple line plot, that is, a line connecting the values of the time series (y-axis) over time (x-axis). We can use this plot to quickly identify potential issues with the data and see if there are any prevailing patterns.

In this recipe, we will show the easiest way to create a line plot. To do so, we will download Microsoft's stock prices from 2020.

How to do it...

Execute the following steps to download, preprocess, and plot Microsoft's stock prices and returns series:

1. Import the libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
```

2. Download Microsoft's stock prices from 2020 and calculate simple returns:

```
df = yf.download("MSFT",
                 start="2020-01-01",
                 end="2020-12-31",
                 auto_adjust=False,
                 progress=False)
df["simple_rtn"] = df["Adj Close"].pct_change()
df = df.dropna()
```

We dropped the NaNs introduced by calculating the percentage change. This only affects the first row.

3. Plot the adjusted close prices:

```
df["Adj Close"].plot(title="MSFT stock in 2020")
```

Executing the one-liner above generates the following plot:

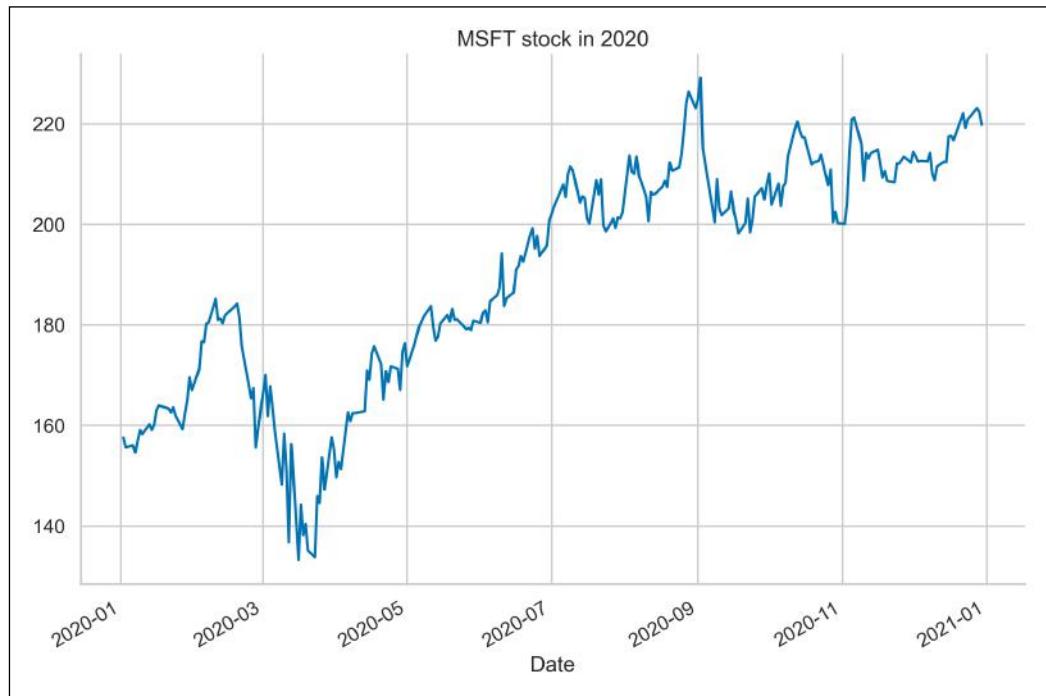


Figure 3.1: Microsoft's adjusted stock price in 2020

Plot the adjusted close prices and simple returns in one plot:

```
(  
    df[["Adj Close", "simple_rtn"]]  
    .plot(subplots=True, sharex=True,  
          title="MSFT stock in 2020")  
)
```

Running the code generates the following plot:

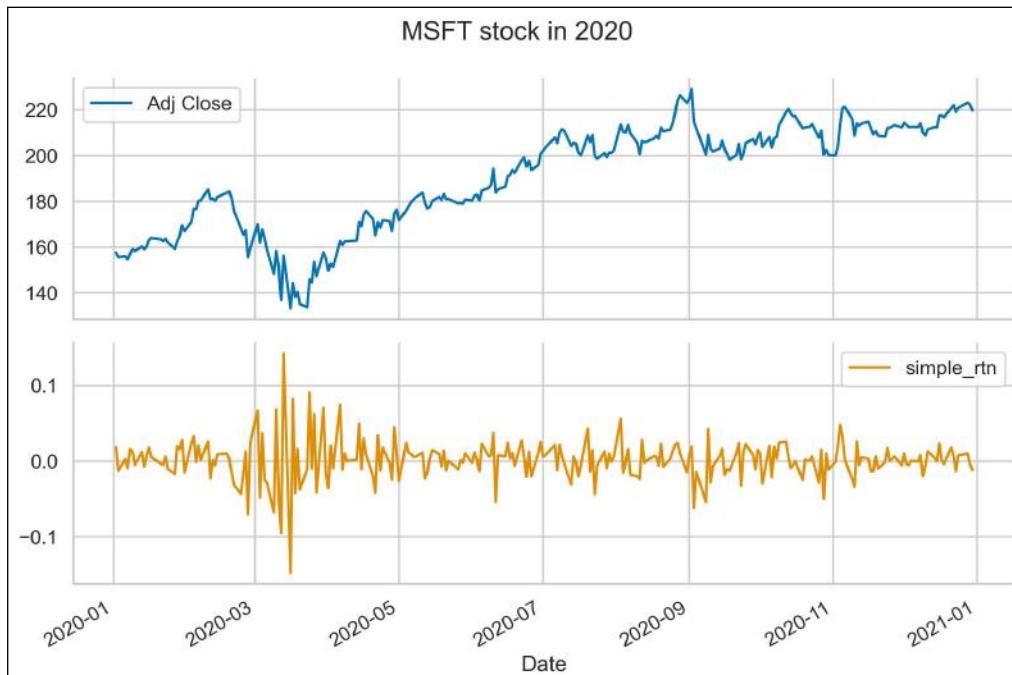


Figure 3.2: Microsoft's adjusted stock price and simple returns in 2020

In *Figure 3.2*, we can clearly see that the dip in early 2020—caused by the start of the COVID-19 pandemic—resulted in increased volatility (variability) of returns. We will get more familiar with volatility in the next chapters.

How it works...

After importing the libraries, we downloaded Microsoft stock prices from 2020 and calculated simple returns using the adjusted close price.

Then, we used the `plot` method of a pandas DataFrame to quickly create a line plot. The only argument we specified was the plot's title. Something to keep in mind is that we used the `plot` method only after subsetting a single column from the DataFrame (which is effectively a `pd.Series` object) and the dates were automatically picked up for the x-axis as they were the index of the DataFrame/Series.

We could have also used a more explicit notation to create the very same plot:

```
df.plot.line(y="Adj Close", title="MSFT stock in 2020")
```

The `plot` method is by no means restricted to creating line charts (which are the default). We can also create histograms, bar charts, scatterplots, pie charts, and so on. To select those, we need to specify the `kind` argument with a corresponding type of plot. Please bear in mind that for some kinds of plots (like the scatterplot), we might need to explicitly provide the values for both axes.



In Step 4, we created a plot consisting of two subplots. We first selected the columns of interest (prices and returns) and then used the `plot` method while specifying that we want to create subplots and that they should share the x-axis.

There's more...

There are many more interesting things worth mentioning about creating line plots, however, we will only cover the following two, as they might be the most useful in practice.

First, we can create a similar plot to the previous one using `matplotlib`'s object-oriented interface:

```
fig, ax = plt.subplots(2, 1, sharex=True)

df["Adj Close"].plot(ax=ax[0])
ax[0].set(title="MSFT time series",
           ylabel="Stock price ($)")

df["simple rtn"].plot(ax=ax[1])
ax[1].set(ylabel="Return (%)")
plt.show()
```

Running the code generates the following plot:

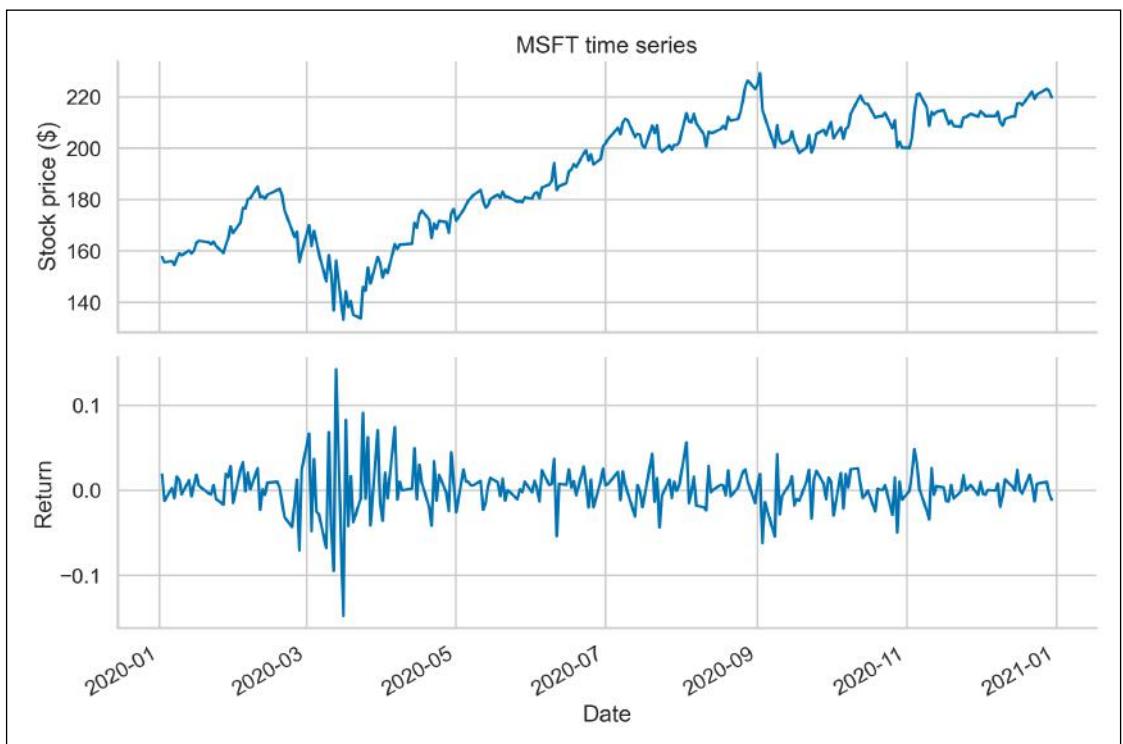


Figure 3.3: Microsoft's adjusted stock price and simple returns in 2020

While it is very similar to the previous plot, we have included some more details on it, such as y-axis labels.

One thing that is quite important here, and which will also be useful later on, is the object-oriented interface of `matplotlib`. While calling `plt.subplots`, we indicated we want to create two subplots in a single column, and we also specified that they will be sharing the x-axis. But what is really crucial is the output of the function, that is:

- An instance of the `Figure` class called `fig`. We can think of it as the container for our plots.
- An instance of the `Axes` class called `ax` (not to be confused with the plot's x- and y-axes). These are all the requested subplots. In our case, we have two of them.

Figure 3.4 illustrates the relationship between a figure and the axes:

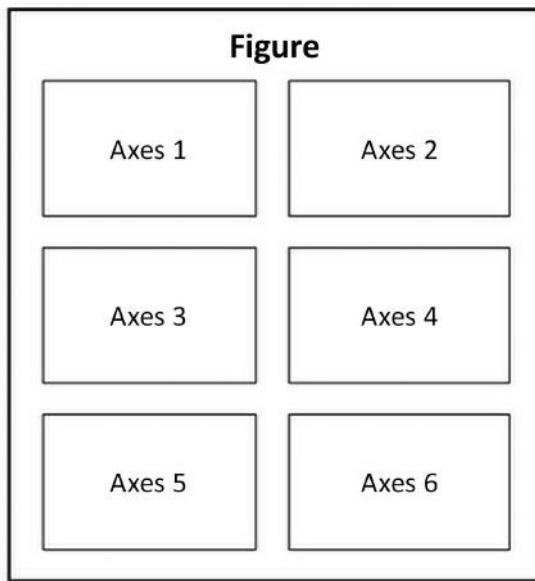


Figure 3.4: The relationship between matplotlib's figure and axes

With any figure, we can have an arbitrary number of subplots arranged in some form of a matrix. We can also create more complex configurations, in which the top row might be a single wide subplot, while the bottom row might be composed of two smaller subplots, each half the size of the large one.

While building the plot above, we have still used the `plot` method of a pandas `DataFrame`. The difference is that we have explicitly specified where in the figure we would like to place the subplots. We have done that by providing the `ax` argument. Naturally, we could have also used `matplotlib`'s functions for creating the plot, but we wanted to save a few lines of code.

The second thing worth mentioning is that we can change the plotting backend of pandas to some other libraries, like `plotly`. We can do so using the following snippet:

```
df["Adj Close"].plot(title="MSFT stock in 2020", backend="plotly")
```

Running the code generates the following interactive plot:

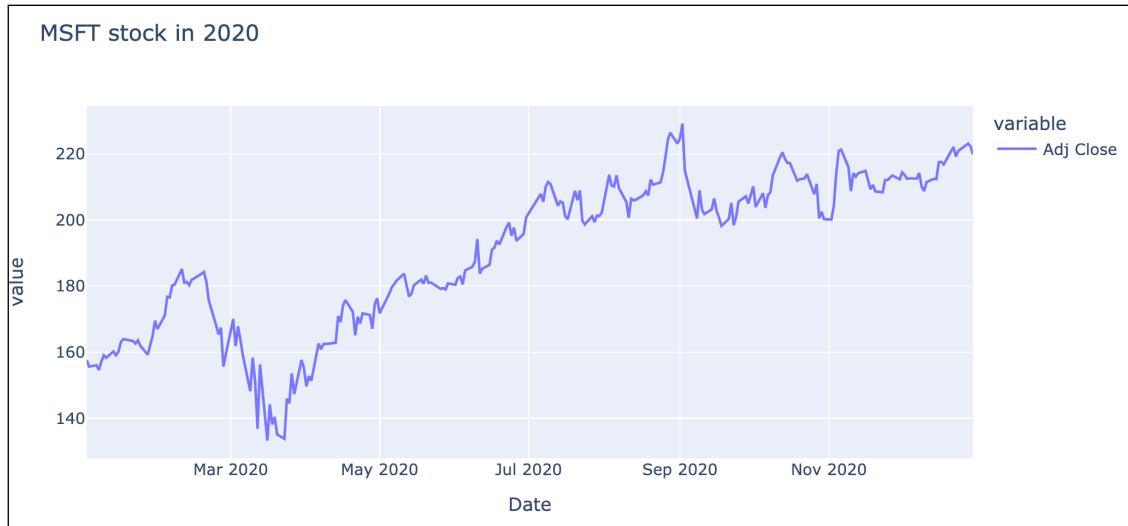


Figure 3.5: Microsoft’s adjusted stock price in 2020, visualized using `plotly`

Unfortunately, the advantages of using the `plotly` backend are not visible in print. In the notebook, you can hover over the plot to see the exact values (and any other information we include in the tooltip), zoom in on particular periods, filter the lines (if there are multiple), and much more. Please see the accompanying notebook (available on GitHub) to test out the interactive features of the visualization.

While changing the backend of the `plot` method, we should be aware of two things:

- We need to have the corresponding libraries installed.
- Some backends have issues with certain functionalities of the `plot` method, most notably the `subplots` argument.

 To generate the previous plot, we specified the plotting backend while creating the plot. That means the next plot we create without specifying it explicitly will be created using the default backend (`matplotlib`). We can use the following snippet to change the plotting backend for our entire session/notebook:
`pd.options.plotting.backend = "plotly".`

See also

<https://matplotlib.org/stable/index.html>—`matplotlib`'s documentation is a treasure trove of information about the library. Most notably, it contains useful tutorials and hints on how to create custom visualizations.

Visualizing seasonal patterns

As we will learn in *Chapter 6, Time Series Analysis and Forecasting*, seasonality plays a very important role in time series analysis. By **seasonality**, we mean the presence of patterns that occur at regular intervals (shorter than a year). For example, imagine the sales of ice cream, which most likely experience a peak in the summer months, while the sales decrease in winter. And such patterns can be seen year over year. We show how to use the line plot with a slight twist to efficiently investigate such patterns.

In this recipe, we will visually investigate seasonal patterns in the US unemployment rate from the years 2014-2019.

How to do it...

Execute the following steps to create a line plot showing seasonal patterns:

1. Import the libraries and authenticate:

```
import pandas as pd
import nasdaqdatalink
import seaborn as sns

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

2. Download and display unemployment data from Nasdaq Data Link:

```
df = (
    nasdaqdatalink.get(dataset="FRED/UNRATENSA",
                       start_date="2014-01-01",
                       end_date="2019-12-31")
    .rename(columns={"Value": "unemp_rate"})
)
df.plot(title="Unemployment rate in years 2014-2019")
```

Running the code generates the following plot:

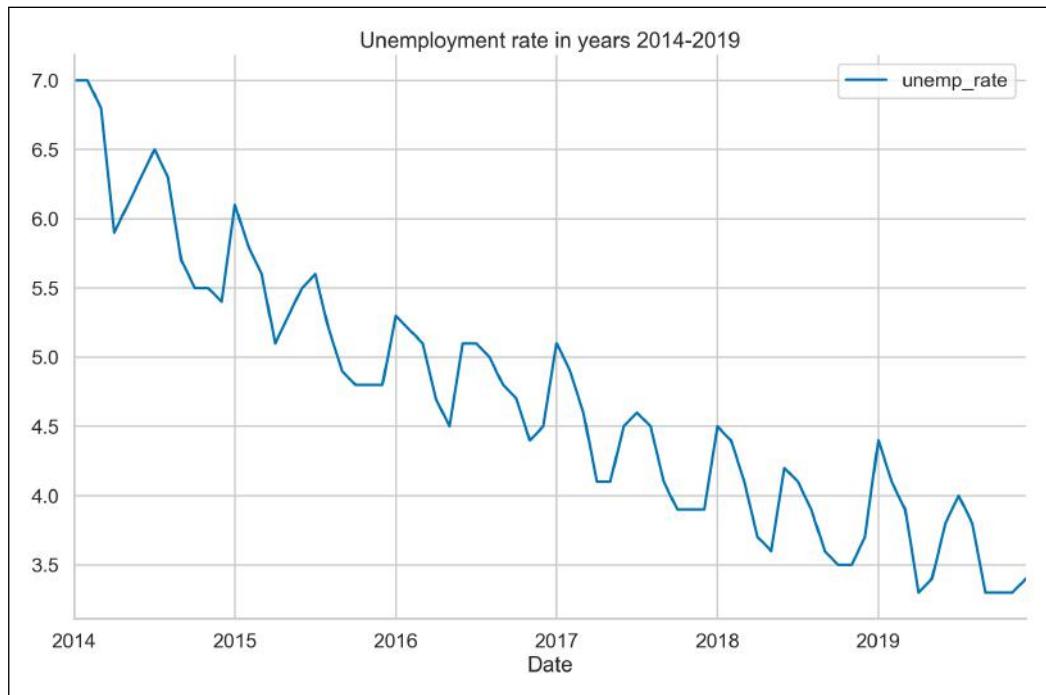


Figure 3.6: Unemployment rate (US) in the years 2014 to 2019

The unemployment rate expresses the number of unemployed as a percentage of the labor force. The values are not adjusted for seasonality, so we can try to spot some patterns.

In *Figure 3.6*, we can already spot some seasonal (repeating) patterns, for example, each year unemployment seems to be highest in January.

3. Create new columns with year and month:

```
df["year"] = df.index.year  
df["month"] = df.index.strftime("%b")
```

4. Create the seasonal plot:

```

sns.lineplot(data=df,
              x="month",
              y="unemp_rate",
              hue="year",
              style="year",
              legend="full",
              palette="colorblind")
plt.title("Unemployment rate - Seasonal plot")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2)

```

Running the code results in the following plot:

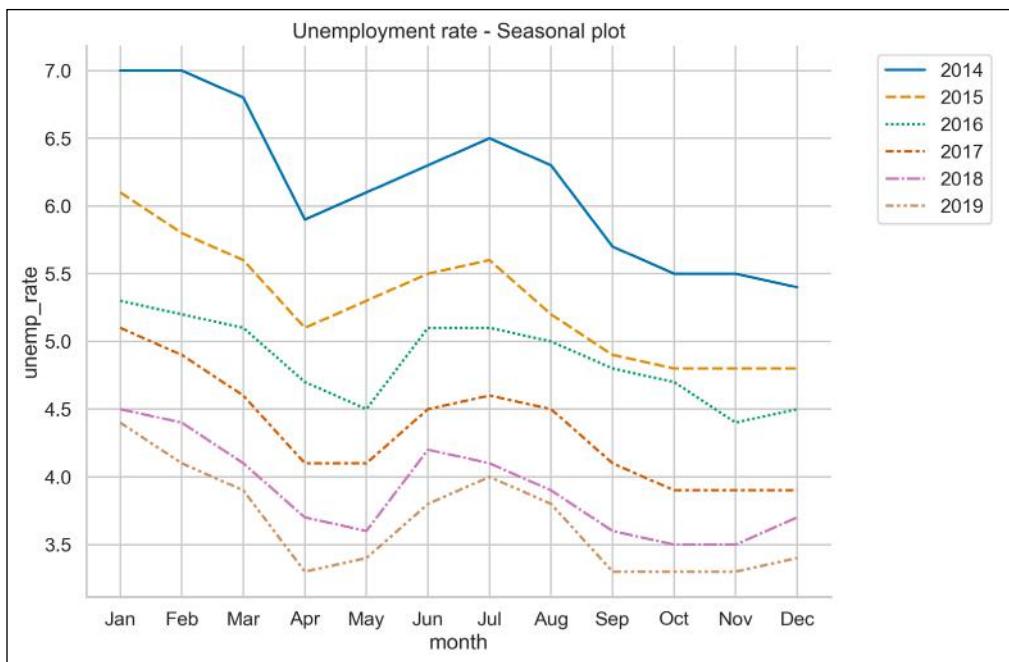


Figure 3.7: Seasonal plot of the unemployment rate

By displaying each year's unemployment rate over the months, we can clearly see some seasonal patterns. For example, the highest unemployment can be observed in January, while the lowest is in December. Also, there seems to be a consistent increase in unemployment over the summer months.

How it works...

In the first step, we imported the libraries and authenticated with Nasdaq Data Link. In the second step, we downloaded the unemployment data from the years 2014-2019. For convenience, we renamed the Value column to unemp_rate.

In Step 3, we created two new columns, in which we extracted the year and the name of the month from the index (encoded as DatetimeIndex).

In the last step, we used the `sns.lineplot` function to create the seasonal line plot. We specified that we want to use the months on the x-axis and that we will plot each year as a separate line (using the `hue` argument).



We can create such plots using other libraries as well. We used `seaborn` (which is a wrapper around `matplotlib`) to showcase the library. In general, it is recommended to use `seaborn` when you would like to include some statistical information on the plot as well, for example, to plot the line of best fit on a scatterplot.

There's more...

We have already investigated the simplest way to investigate seasonality on a plot. In this part, we will also go over some alternative visualizations that can reveal additional information about seasonal patterns.

1. Import the libraries:

```
from statsmodels.graphics.tsaplots import month_plot, quarter_plot
import plotly.express as px
```

2. Create a month plot:

```
month_plot(df["unemp_rate"], ylabel="Unemployment rate (%)")
plt.title("Unemployment rate - Month plot")
```

Running the code produces the following plot:

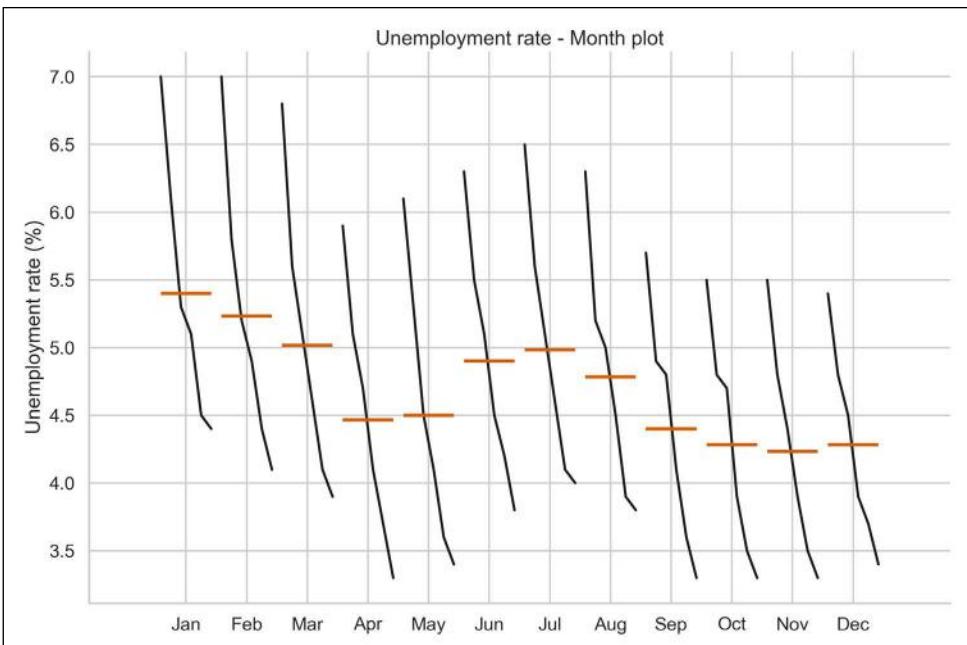


Figure 3.8: The month plot of the unemployment rate

A month plot is a simple yet informative visualization. For each month, it plots a separate line showing how the unemployment rate changed over time (while not showing the time points explicitly). Additionally, the red horizontal lines show the average values in those months.

We can draw some conclusions from analyzing *Figure 3.8*:

- By looking at the average values, we can see the pattern we have described before – the highest values are observed in January, then the unemployment rate decreases, only to bounce back over the summer months and then continue decreasing until the end of the year.
- Over the years, the unemployment rate decreased; however, in 2019, the decrease seems to be smaller than in the previous years. We can see this by looking at the different angles of the lines in July and August.

3. Create a quarter plot:

```
quarter_plot(df["unemp_rate"].resample("Q").mean(),
            ylabel="Unemployment rate (%)")
plt.title("Unemployment rate - Quarter plot")
```

Running the code produces the following figure:

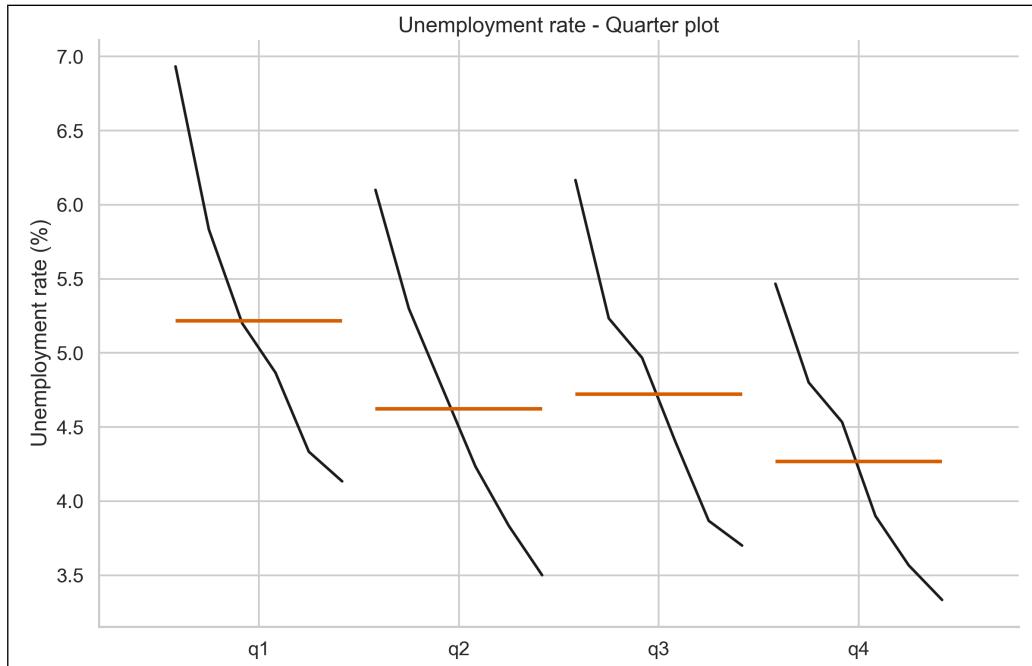


Figure 3.9: The quarter plot of the unemployment rate

The **quarter plot** is very similar to the month plot, the only difference being that we use quarters instead of months on the x-axis. To arrive at this plot, we had to resample the monthly unemployment rate by taking each quarter's average value. We could have taken the last value as well.

4. Create a polar seasonal plot using `plotly.express`:

```
fig = px.line_polar(  
    df, r="unemp_rate", theta="month",  
    color="year", line_close=True,  
    title="Unemployment rate - Polar seasonal plot",  
    width=600, height=500,  
    range_r=[3, 7]  
)  
fig.show()
```

Running the code produces the following interactive plot:

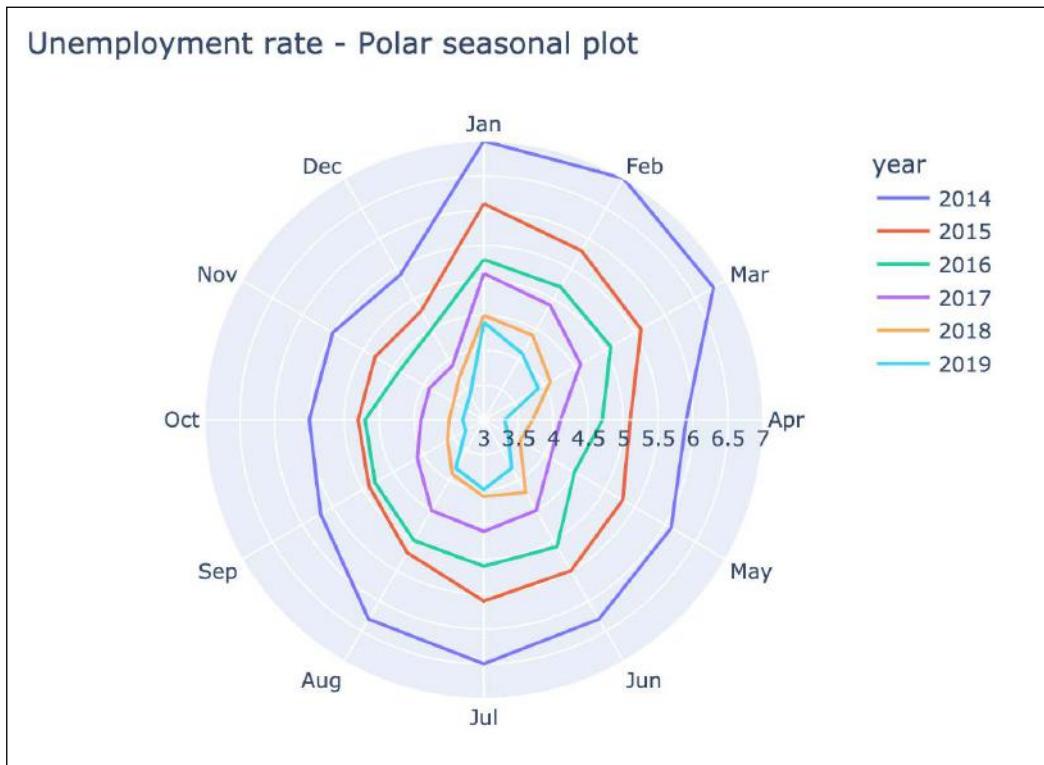


Figure 3.10: Polar seasonal plot of the unemployment rate

Lastly, we created a variation of the seasonal plot in which we plotted the lines on the polar coordinate plane. It means that the polar chart visualizes the data along radial and angular axes. We have manually capped the radial range by setting `range_r=[3, 7]`. Otherwise, the plot would have started at 0 and it would be harder to see any difference between the lines.

The conclusions we can draw are similar to those from a normal seasonal plot, however, it might take a while to get used to this representation. For example, by looking at the year 2014, we immediately see that unemployment is highest in the first quarter of the year.

Creating interactive visualizations

In the first recipe, we gave a short preview of creating interactive visualizations in Python. In this recipe, we will show how to create interactive line plots using three different libraries: `cufflinks`, `plotly`, and `bokeh`. Naturally, these are not the only available libraries for interactive visualizations. Another popular one you might want to investigate further is `altair`.

The `plotly` library is built on top of `d3.js` (a JavaScript library used for creating interactive visualizations in web browsers) and is known for creating high-quality plots with a significant degree of interactivity (inspecting values of observations, viewing tooltips of a given point, zooming in, and so on). Plotly is also the company responsible for developing this library and it provides hosting for our visualizations. We can create an infinite number of offline visualizations and a few free ones to share online (with a limited number of views per day).

`cufflinks` is a wrapper library built on top of `plotly`. It was released before `plotly.express` was introduced as part of the `plotly` framework. The main advantages of `cufflinks` are:

- It makes the plotting much easier than pure `plotly`.
- It enables us to create the `plotly` visualizations directly on top of `pandas` DataFrames.
- It contains a selection of interesting specialized visualizations, including a special class for quantitative finance (which we will cover in the next recipe).

Lastly, `bokeh` is another library for creating interactive visualizations, aiming particularly for modern web browsers. Using `bokeh`, we can create beautiful interactive graphics, from simple line plots to complex interactive dashboards with streaming datasets. The visualizations of `bokeh` are powered by JavaScript, but actual knowledge of JavaScript is not explicitly required for creating the visualizations.

In this recipe, we will create a few interactive line plots using Microsoft's stock price from 2020.

How to do it...

Execute the following steps to download Microsoft's stock prices and create interactive visualizations:

1. Import the libraries and initialize the notebook display:

```
import pandas as pd
import yfinance as yf

import cufflinks as cf
from plotly.offline import iplot, init_notebook_mode
import plotly.express as px
import pandas_bokeh

cf.go_offline()
pandas_bokeh.output_notebook()
```

2. Download Microsoft's stock prices from 2020 and calculate simple returns:

```
df = yf.download("MSFT",
                  start="2020-01-01",
                  end="2020-12-31",
                  auto_adjust=False,
                  progress=False)

df["simple rtn"] = df["Adj Close"].pct_change()
df = df.loc[:, ["Adj Close", "simple rtn"]].dropna()
df = df.dropna()
```

3. Create the plot using cufflinks:

```
df.iplot(subplots=True, shape=(2,1),
          shared_xaxes=True,
          title="MSFT time series")
```

Running the code creates the following plot:



Figure 3.11: Example of time series visualization using cufflinks

With the plots generated using cufflinks and plotly, we can hover over the line to see the tooltip containing the date of the observation and the exact value (or any other available information). We can also select a part of the plot that we would like to zoom in on for easier analysis.

4. Create the plot using bokeh:

```
df["Adj Close"].plot_bokeh(kind="line",
                           rangetool=True,
                           title="MSFT time series")
```

Executing the code generates the following plot:

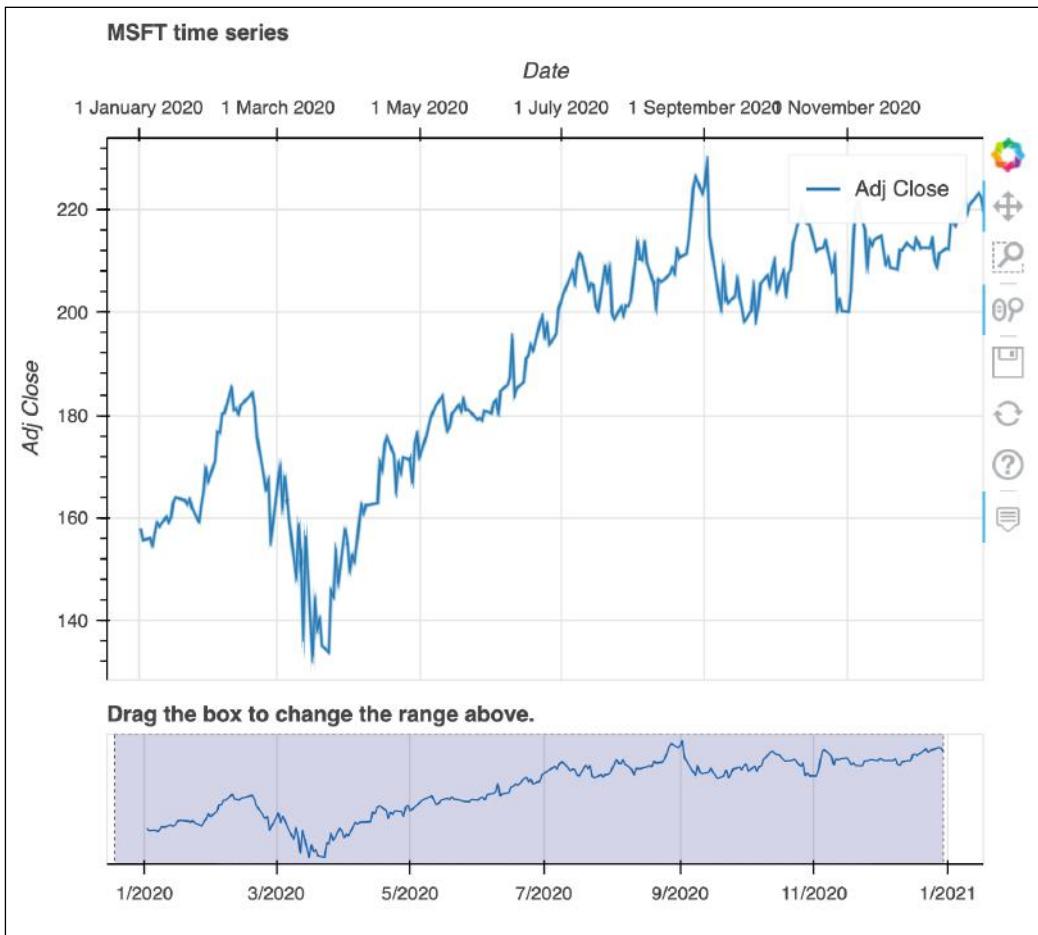


Figure 3.12: Microsoft's adjusted stock prices visualized using Bokeh

By default, the bokeh plot comes not only with the tooltip and zooming functionalities, but also the range slider. We can use it to easily narrow down the range of dates that we would like to see in the plot.

5. Create the plot using `plotly.express`:

```
fig = px.line(data_frame=df,
               y="Adj Close",
               title="MSFT time series")
fig.show()
```

Running the code results in the following visualization:

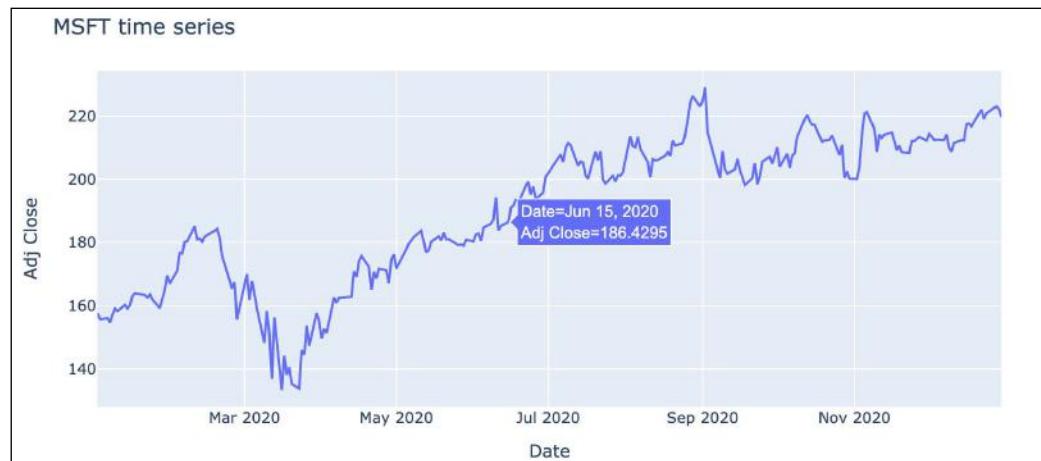


Figure 3.13: Example of time series visualization using plotly

In *Figure 3.13*, you can see an example of the interactive tooltip, which is useful for identifying particular observations within the analyzed time series.

How it works...

In the first step, we imported the libraries and initialized the notebook display for bokeh and the offline mode for cufflinks. Then, we downloaded Microsoft's stock prices from 2020, calculated simple returns using the adjusted close price, and only kept those two columns for further plotting.

In the third step, we created the first interactive visualization using cufflinks. As mentioned in the introduction, thanks to cufflinks, we can use the `iplot` method directly on top of the pandas DataFrame. It works similarly to the original `plot` method. Here, we indicated that we wanted to create subplots in one column, sharing the x-axis. The library handled the rest and created a nice and interactive visualization.

In *Step 4*, we created a line plot using bokeh. We did not use the pure bokeh library, but an official wrapper around pandas—`pandas_bokeh`. Thanks to it, we could access the `plot_bokeh` method directly on top of the pandas DataFrame to simplify the process of creating the plot.

Lastly, we used the `plotly.express` framework, which is now officially part of the `plotly` library (it used to be a standalone library). Using the `px.line` function, we can easily create a simple, yet interactive line plot.

There's more...

While using the visualizations to tell a story or presenting the outputs of our analyses to stakeholders or a non-technical audience, there are a few techniques that might improve the plot's ability to convey a given message. Annotations are one of those techniques and we can easily add them to the plots generated with `plotly` (we can do so with other libraries as well).

We show the required steps below:

1. Import the libraries:

```
from datetime import date
```

2. Define the annotations for the plotly plot:

```
selected_date_1 = date(2020, 2, 19)
selected_date_2 = date(2020, 3, 23)

first_annotation = {
    "x": selected_date_1,
    "y": df.query(f"index == '{selected_date_1}'")["Adj Close"].squeeze(),
    "arrowhead": 5,
    "text": "COVID decline starting",
    "font": {"size": 15, "color": "red"},
}

second_annotation = {
    "x": selected_date_2,
    "y": df.query(f"index == '{selected_date_2}'")["Adj Close"].squeeze(),
    "arrowhead": 5,
    "text": "COVID recovery starting",
    "font": {"size": 15, "color": "green"},
    "ax": 150,
    "ay": 10
}
```

The dictionaries contain a few elements that might be worthwhile to explain:

- `x/y`—The location of the annotation on the x- and y-axes respectively
- `text`—The text of the annotation
- `font`—The font's formatting
- `arrowhead`—The shape of the arrowhead we want to use
- `ax/ay`—The offset along the x- and y-axes from the indicated point

We frequently use the offset to make sure that the annotations are not overlapping with each other or with other elements of the plot.

After defining the annotations, we can simply add them to the plot.

3. Update the layout of the plot and show it:

```
fig.update_layout(  
    {"annotations": [first_annotation, second_annotation]}  
)  
fig.show()
```

Running the snippet generates the following plot:



Figure 3.14: Time series visualization with added annotations

Using the annotations, we have marked the dates when the market started to decline due to the COVID-19 pandemic, as well as when it started to recover and rise again. The dates used for annotations were selected simply by viewing the plot.

See also

- <https://bokeh.org/>—For more information about bokeh.
- <https://altair-viz.github.io/>—You can also inspect altair, another popular Python library for interactive visualizations.
- <https://plotly.com/python/>—plotly's Python documentation. The library is also available for other programming languages such as R, MATLAB, or Julia.

Creating a candlestick chart

A candlestick chart is a type of financial graph, used to describe a given security's price movements. A single candlestick (typically corresponding to one day, but a different frequency is possible) combines the open, high, low, and close (OHLC) prices.

The elements of a bullish candlestick (where the close price in a given time period is higher than the open price) are presented in *Figure 3.15*:

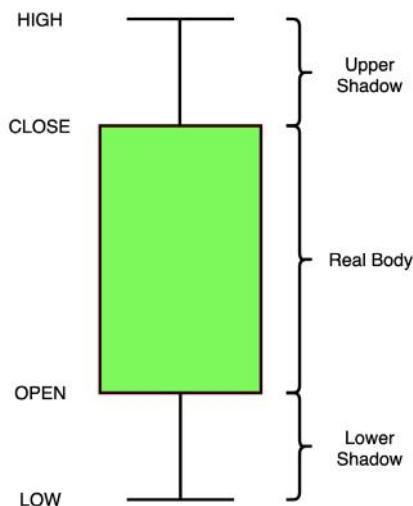


Figure 3.15: Diagram of a bullish candlestick

For a bearish candlestick, we should swap the positions of the open and close prices. Typically, we would also change the candle's color to red.

In comparison to the plots introduced in the previous recipes, candlestick charts convey much more information than a simple line plot of the adjusted close price. That is why they are often used in real trading platforms, and traders use them for identifying patterns and making trading decisions.

In this recipe, we also add moving average lines (which are one of the most basic technical indicators), as well as bar charts representing volume.

Getting ready

In this recipe, we will download Twitter's (adjusted) stock prices for the year 2018. We will use Yahoo Finance to download the data, as described in *Chapter 1, Acquiring Financial Data*. Follow these steps to get the data for plotting:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
```

2. Download the adjusted prices:

```
df = yf.download("TWTR",
                  start="2018-01-01",
                  end="2018-12-31",
                  progress=False,
                  auto_adjust=True)
```

How to do it...

Execute the following steps to create an interactive candlestick chart:

1. Import the libraries:

```
import cufflinks as cf
from plotly.offline import iplot

cf.go_offline()
```

2. Create the candlestick chart using Twitter's stock prices:

```
qf = cf.QuantFig(
    df, title="Twitter's Stock Price",
    legend="top", name="Twitter's stock prices in 2018"
)
```

3. Add volume and moving averages to the figure:

```
qf.add_volume()
qf.add_sma(periods=20, column="Close", color="red")
qf.add_ema(periods=20, color="green")
```

4. Display the plot:

```
qf.iplot()
```

We can observe the following plot (it is interactive in the notebook):



Figure 3.16: Candlestick plot of Twitter's stock prices in 2018

In the plot, we can see that the **exponential moving average (EMA)** adapts to the changes in prices much faster than the **simple moving average (SMA)**. Some discontinuities in the chart are caused by the fact that we are using daily data, and there is no data for weekends/bank holidays.

How it works...

In the first step, we imported the required libraries and indicated that we wanted to use the offline mode of `cufflinks` and `plotly`.



As an alternative to running `cf.go_offline()` every time, we can also modify the settings to always use the offline mode by running `cf.set_config_file(offline=True)`. We can then view the settings using `cf.get_config_file()`.

In *Step 2*, we created an instance of a `QuantFig` object by passing a `DataFrame` containing the input data, as well as some arguments for the title and legend's position. We could have created a simple candlestick chart by running the `iplot` method of `QuantFig` immediately afterward.

In *Step 3*, we added two moving average lines by using the `add_sma/add_ema` methods. We decided to consider 20 periods (days, in this case). By default, the averages are calculated using the `close` column, however, we can change this by providing the `column` argument.

The difference between the two moving averages is that the exponential one puts more weight on recent prices. By doing so, it is more responsive to new information and reacts faster to any changes in the general trend.

Lastly, we displayed the plot using the `iplot` method.

There's more...

As mentioned in the chapter's introduction, there are often multiple ways we can do the same task in Python, often using different libraries. We will also show how to create candlestick charts using pure `plotly` (in case you do not want to use a wrapper library such as `cufflinks`) and `mplfinance`, a standalone expansion to `matplotlib` dedicated to plotting financial data:

1. Import the libraries:

```
import plotly.graph_objects as go
import mplfinance as mpf
```

2. Create a candlestick chart using pure `plotly`:

```
fig = go.Figure(data=
    go.Candlestick(x=df.index,
                    open=df["Open"],
                    high=df["High"],
                    low=df["Low"],
                    close=df["Close"])
)
```

```
fig.update_layout(  
    title="Twitter's stock prices in 2018",  
    yaxis_title="Price ($)"  
)  
  
fig.show()
```

Running the snippet results in the following plot:



Figure 3.17: An example of a candlestick chart generated using plotly

The code is a bit lengthy, but in reality, it is quite straightforward. We needed to pass an object of class `go.Candlestick` as the `data` argument for the figure defined using `go.Figure`. Then, we just added the title and the label for the y-axis using the `update_layout` method.

What is convenient about the `plotly` implementation of the candlestick chart is that it comes with a range slider, which we can use to interactively narrow down the displayed candlesticks to the period that we want to investigate in more detail.

3. Create a candlestick chart using `mplfinance`:

```
mpf.plot(df, type="candle",  
        mav=(10, 20),  
        volume=True,  
        style="yahoo",  
        title="Twitter's stock prices in 2018",  
        figsize=(8, 4))
```

Running the code generated the following plot:

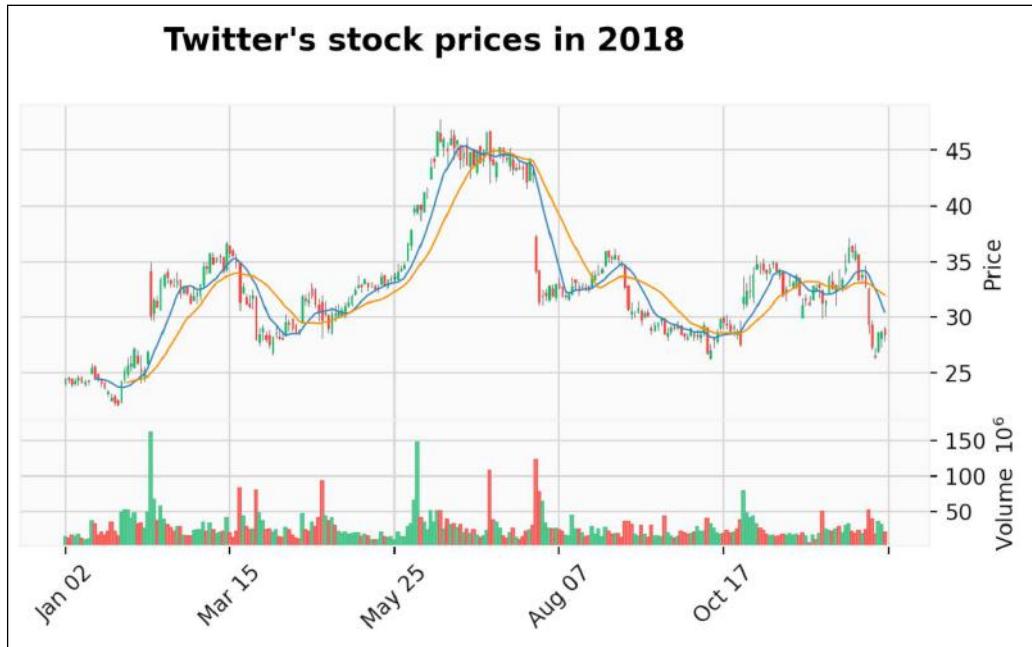


Figure 3.18: An example of a candlestick chart generated using mplfinance

We used the `mav` argument to indicate we wanted to create two moving averages, 10- and 20-day ones. Unfortunately, at this moment, it is not possible to add exponential variants. However, we can add additional plots to the figure using the `mpf.make_addplot` helper function. We also indicated that we wanted to use a style resembling the one used by Yahoo Finance.



You can use the command `mpf.available_styles()` to display all the available styles.

See also

Some useful references:

- <https://github.com/santosjorge/cufflinks>—The GitHub repository of cufflinks
- https://github.com/santosjorge/cufflinks/blob/master/cufflinks/quant_figure.py—The source code of cufflinks might be helpful for getting more information on the available methods (different indicators and settings)
- <https://github.com/matplotlib/mplfinance>—The GitHub repository of `mplfinance`
- <https://github.com/matplotlib/mplfinance/blob/master/examples/addplot.ipynb>—A Notebook with examples of how to add extra information to plots generated with `mplfinance`

Summary

In this chapter, we have covered various ways of visualizing financial (and not only) time series. Plotting the data is very helpful in getting familiar with the analyzed time series. We can identify some patterns (for example, trends or changepoints) that we might subsequently want to confirm with statistical tests. Visualizing data can also help to spot some outliers (extreme values) in our series. This brings us to the topic of the next chapter, that is, automatic pattern identification and outlier detection.

4

Exploring Financial Time Series Data

In the previous chapters, we learned how to preprocess and visually explore financial time series data. This time, we will use algorithms and/or statistical tests to automatically identify potential issues (like outliers) and analyze the data for the existence of trends or other patterns (for example, mean-reversion).

We will also dive deeper into the stylized facts of asset returns. Together with outlier detection, those recipes are particularly important when working with financial data. When we want to build models/strategies based on asset prices, we have to make sure that they can accurately capture the dynamics of the returns.

Having said that, most of the techniques described in this chapter are not restricted only to financial time series and can be effectively used in other domains as well.

In this chapter, we will cover the following recipes:

- Outlier detection using rolling statistics
- Outlier detection with the Hampel filter
- Detecting changepoints in time series
- Detecting trends in time series
- Detecting patterns in a time series using the Hurst exponent
- Investigating stylized facts of asset returns

Outlier detection using rolling statistics

While working with any kind of data, we often encounter observations that are significantly different from the majority, that is, **outliers**. In the financial domain, they can be the result of a wrong price, something major happening in the financial markets, or an error in the data processing pipeline. Many machine learning algorithms and statistical approaches can be heavily influenced by outliers, leading to incorrect/biased results. That is why we should identify and handle the outliers before creating any models.



In this chapter, we will focus on point anomaly detection, that is, investigating whether a given observation stands out in comparison to the other ones. There are different sets of algorithms that can identify entire sequences of data as anomalous.

In this recipe, we cover a relatively simple, filter-like approach to detect outliers based on the rolling average and standard deviation. We will use Tesla's stock prices from the years 2019 to 2020.

How to do it...

Execute the following steps to detect outliers using the rolling statistics and mark them on the plot:

1. Import the libraries:

```
import pandas as pd  
import yfinance as yf
```

2. Download Tesla's stock prices from 2019 to 2020 and calculate simple returns:

```
df = yf.download("TSLA",  
                 start="2019-01-01",  
                 end="2020-12-31",  
                 progress=False)  
  
df["rtn"] = df["Adj Close"].pct_change()  
df = df[["rtn"]].copy()
```

3. Calculate the 21-day rolling mean and standard deviation:

```
df_rolling = df[["rtn"]].rolling(window=21) \  
            .agg(["mean", "std"])  
df_rolling.columns = df_rolling.columns.droplevel()
```

4. Join the rolling data back to the initial DataFrame:

```
df = df.join(df_rolling)
```

5. Calculate the upper and lower thresholds:

```
N_SIGMAS = 3  
df["upper"] = df["mean"] + N_SIGMAS * df["std"]  
df["lower"] = df["mean"] - N_SIGMAS * df["std"]
```

6. Identify the outliers using the previously calculated thresholds:

```
df["outlier"] = (  
    (df["rtn"] > df["upper"]) | (df["rtn"] < df["lower"])  
)
```

7. Plot the returns together with the thresholds and mark the outliers:

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

df[["rtn", "upper", "lower"]].plot(ax=ax)
ax.scatter(df.loc[df["outlier"]].index,
           df.loc[df["outlier"], "rtn"],
           color="black", label="outlier")
ax.set_title("Tesla's stock returns")
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))

plt.show()
```

Running the snippet generates the following plot:



Figure 4.1: Outliers identified using the filtering algorithm

In the plot, we can observe outliers marked with a black dot, together with the thresholds used for determining them. One thing to notice is that when there are two large (in absolute terms) returns in the vicinity of one another, the algorithm identifies the first one as an outlier and the second one as a regular observation. This might be due to the fact that the first outlier enters the rolling window and affects the moving average/standard deviation. We can observe a situation like that in the first quarter of 2020.



We should also be aware of the so-called ghost effect. When a single outlier enters the rolling window, it inflates the values of the rolling statistics for as long as it remains in the rolling window.

How it works...

After importing the libraries, we downloaded Tesla's stock prices, calculated the returns, and only kept a single column — the one with the returns — for further analysis.

To identify the outliers, we started by calculating the moving statistics using a 21-day rolling window. We used 21 as this is the average number of trading days in a month, and in this example, we work with daily data. However, we can choose different values, and then the moving average will react more quickly/slowly to changes. We can also use the (exponentially) weighted moving average if we find it more meaningful for our particular case. To implement the moving metrics, we used the combination of the `rolling` and `agg` methods of a pandas DataFrame. After calculating the statistics, we dropped one level of the MultiIndex to simplify the analysis.



When applying the rolling window, we used the previous 21 observations to calculate the statistics. So the first value is available for the 22nd row of the DataFrame. By using this approach, we do not “leak” future information into the algorithm. However, there might be cases when we do not really mind such leakage. In those cases, we might want to use centered windows. Then, using the same window size, we would consider the past 10 observations, the current one, and the next 10 future data points. To do so, we can use the `center` argument of the `rolling` method.

In *Step 4*, we joined the rolling statistics back to the original DataFrame. Then, we created additional columns containing the upper and lower decision thresholds. We decided to use 3 standard deviations above/below the rolling average as the boundaries. Any observation lying beyond them was considered an outlier. We should keep in mind that the logic of the filtering algorithm is based on the assumption of the stock returns being normally distributed. Later in the chapter, we will see that this assumption does not hold empirically. We coded that condition as a separate column in *Step 6*.

In the last step, we visualized the returns series, along with the upper/lower decision thresholds, and marked the outliers with a black dot. To make the plot more readable, we moved the legend outside of the plotting area.

In real-life cases, we should not only identify the outliers but also treat them, for example, by capping them at the maximum/minimum acceptable value, replacing them with interpolated values, or by following any of the other possible approaches.

There's more...

Defining functions

In this recipe, we demonstrated how to carry out all the steps required for identifying the outliers as separate operations on the DataFrame. However, we can quickly encapsulate all of the steps into a single function and make it generic to handle more use cases. You can find an example of how to do so below:

```
def identify_outliers(df, column, window_size, n_sigmas):
    """Function for identifying outliers using rolling statistics"""

    df = df[[column]].copy()
    df_rolling = df.rolling(window=window_size) \
        .agg(["mean", "std"])
    df_rolling.columns = df_rolling.columns.droplevel()
    df = df.join(df_rolling)
    df["upper"] = df["mean"] + n_sigmas * df["std"]
    df["lower"] = df["mean"] - n_sigmas * df["std"]

    return ((df[column] > df["upper"]) | (df[column] < df["lower"]))
```

The function returns a pd.Series containing Boolean flags indicating whether a given observation is an outlier or not. An additional benefit of using a function is that we can easily experiment with using different parameters (such as window sizes and the numbers of standard deviations used for creating the thresholds).

Winsorization

Another popular approach for treating outliers is **winsorization**. It is based on replacing outliers in data to limit their effect on any potential calculations. It's easier to understand winsorization using an example. A 90% winsorization results in replacing the top 5% of values with the 95th percentile. Similarly, the bottom 5% is replaced using the value of the 5th percentile. We can find the corresponding `winsorize` function in the `scipy` library.

Outlier detection with the Hampel filter

We will cover one more algorithm used for outlier detection in time series—the **Hampel filter**. Its goal is to identify and potentially replace outliers in a given series. It uses a centered sliding window of size $2x$ (given x observations before/after) to go over the entire series.

For each of the sliding windows, the algorithm calculates the median and the median absolute deviation (a form of a standard deviation).



For the median absolute deviation to be a consistent estimator for the standard deviation, we have to multiply it by a constant scaling factor k , which is dependent on the distribution. For Gaussian, it is approximately 1.4826.

Similar to the previously covered algorithm, we treat an observation as an outlier if it differs from the window's median by more than a determined number of standard deviations. We can then replace such an observation with the window's median.

We can experiment with the different settings of the algorithm's hyperparameters. For example, a higher standard deviation threshold makes the filter more forgiving, while a lower one results in more data points being classified as outliers.

In this recipe, we will use the Hampel filter to see if any observations in a time series of Tesla's prices from 2019 to 2020 can be considered outliers.

How to do it...

Execute the following steps to identify outliers using the Hampel filter:

1. Import the libraries:

```
import yfinance as yf
from sktime.transformations.series.outlier_detection import HampelFilter
```

2. Download Tesla's stock prices from the years 2019 to 2020 and calculate simple returns:

```
df = yf.download("TSLA",
                 start="2019-01-01",
                 end="2020-12-31",
                 progress=False)
df["rtn"] = df["Adj Close"].pct_change()
```

3. Instantiate the `HampelFilter` class and use it for detecting the outliers:

```
hampel_detector = HampelFilter(window_length=10,
                               return_bool=True)
df["outlier"] = hampel_detector.fit_transform(df["Adj Close"])
```

4. Plot Tesla's stock price and mark the outliers:

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

df[["Adj Close"]].plot(ax=ax)
ax.scatter(df.loc[df["outlier"]].index,
           df.loc[df["outlier"], "Adj Close"],
           color="black", label="outlier")
ax.set_title("Tesla's stock price")
```

```
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))

plt.show()
```

Running the code generates the following plot:

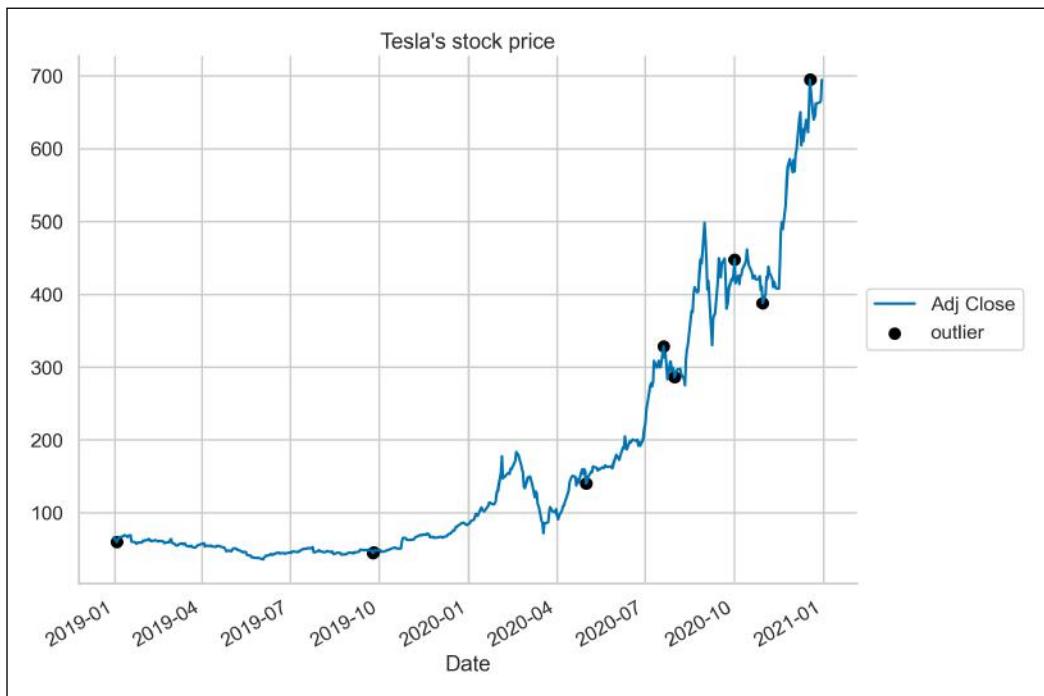


Figure 4.2: Tesla's stock prices and the outliers identified using the Hampel filter

Using the Hampel filter, we identified seven outliers. At first glance, it might be interesting and maybe even a bit counterintuitive that the biggest spike and drop around September 2020 were not detected, but some smaller jumps later on were. We have to remember that this filter uses a centered window, so while looking into the observation at the peak of the spike, the algorithm also looks at the previous and next five observations, which include high values as well.

How it works...

The first two steps are quite standard—we imported the libraries, downloaded the stock prices, and calculated simple returns.

In *Step 3*, we instantiated the object of the `HampelFilter` class. We used the filter's implementation from the `sktime` library, which we will also explore further in *Chapter 7, Machine Learning-Based Approaches to Time Series Forecasting*. We specified that we want to use a window of length 10 (5 observations before and 5 after) and for the filter to return a Boolean flag whether the observation is an outlier or not. The default setting of `return_bool` will return a new series in which the outliers will be replaced with NaNs. That is because the authors of `sktime` suggest using the filter for identifying and removing the outliers, and then using a companion `Imputer` class for filling in the missing values.

sktime uses methods similar to those available in scikit-learn, so we first need to fit the transformer object to the data and then transform it to obtain the flag indicating whether the observation is an outlier. Here, we completed two steps at once by using the fit_transform method applied to the adjusted close price.



Please refer to *Chapter 13, Applied Machine Learning: Identifying Credit Default*, for more information about using scikit-learn's fit/transform API.

In the last step, we plotted the stock price as a line plot and marked the outliers as black dots.

There's more...

For comparison's sake, we can also apply the very same filter to the returns calculated using the adjusted close prices. This way, we can see if the algorithm will identify different observations as outliers:

1. Identify the outliers among the stock returns:

```
df[ "outlier_rtn" ] = hampel_detector.fit_transform(df[ "rtn" ])
```

Because we have already instantiated the HampelFilter, we do not need to do it again. We can just fit it to the new data (returns) and transform it to get the Boolean flag.

2. Plot Tesla's daily returns and mark the outliers:

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

df[ [ "rtn" ] ].plot(ax=ax)
ax.scatter(df.loc[df[ "outlier_rtn" ]].index,
           df.loc[df[ "outlier_rtn" ], "rtn"],
           color="black", label="outlier")
ax.set_title("Tesla's stock returns")
ax.legend(loc="center left", bbox_to_anchor=(1, 0.5))

plt.show()
```

Running the code generates the following plot:

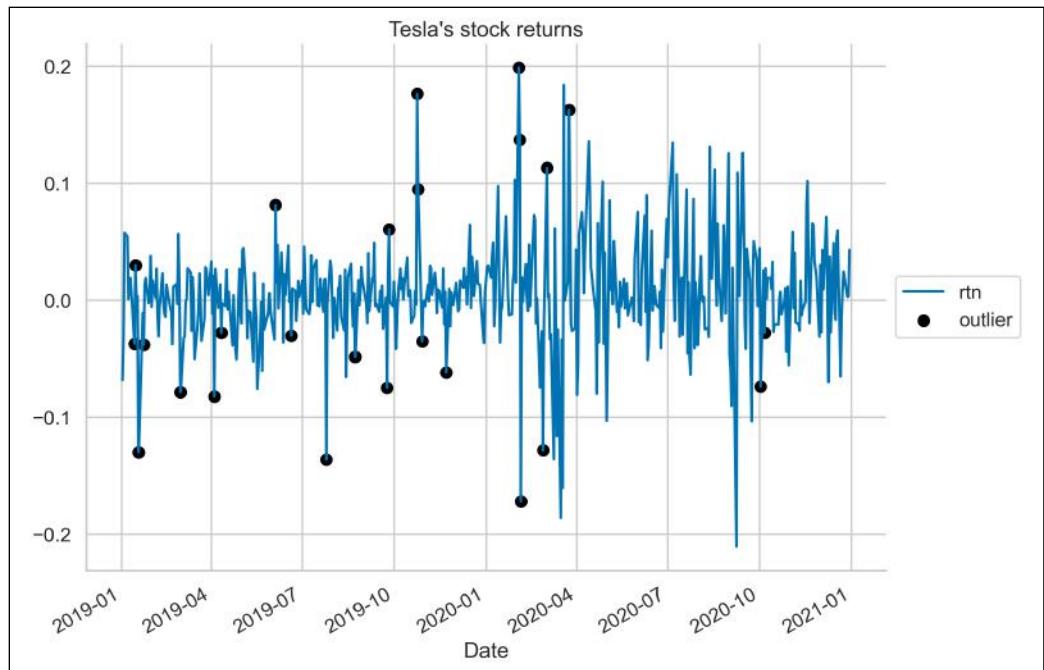


Figure 4.3: Tesla's stock returns and the outliers identified using the Hampel filter

We can immediately see that the algorithm detected more outliers when using the returns instead of the prices.

3. Investigate the overlap in outliers identified for the prices and returns:

```
df.query("outlier == True and outlier_rtn == True")
```

Date	Open	High	Low	Close	Adj Close	Volume	rtn	outlier	outlier_rtn
2019-09-24	48.3	48.4	44.52	44.64	44.64	64457500	-0.07	True	True

Figure 4.4: The date that was identified as an outlier using both prices and returns

There is only a single date that was identified as an outlier based on both prices and returns.

See also

Anomaly/outlier detection is an entire field in data science, and there are numerous approaches to identifying suspicious observations. We have covered two algorithms that are especially suitable for time series problems. However, there are many possible approaches to anomaly detection in general. We will cover outlier detection methods used for data other than time series in *Chapter 13, Applied Machine Learning: Identifying Credit Default*. Some of them can also be used for time series.

Here are a few interesting anomaly/outlier detection libraries:

- <https://github.com/datamllab/tods>
- <https://github.com/zillow/luminaire/>
- <https://github.com/signals-dev/Orion>
- <https://pycaret.org/anomaly-detection/>
- <https://github.com/linkedin/luminol>—a library created by LinkedIn; unfortunately, it is not actively maintained anymore
- <https://github.com/twitter/AnomalyDetection>—this R package (created by Twitter) is quite famous and was ported to Python by some individual contributors

A few more references:

- Hampel F. R. 1974. “The influence curve and its role in robust estimation.” *Journal of the American Statistical Association*, 69: 382–393—a paper introducing the Hampel filter
- <https://www.sktime.org/en/latest/index.html>—documentation of sktime

Detecting changepoints in time series

A **changepoint** can be defined as a point in time when the probability distribution of a process or time series changes, for example, when there is a change to the mean in the series.

In this recipe, we will use the **CUSUM** (cumulative sum) method to detect shifts of the means in a time series. The implementation used in the recipe has two steps:

1. Finding the changepoint—an iterative process is started by first initializing a changepoint in the middle of the given time series. Then, the CUSUM approach is carried out based on the selected point. The following changepoint is located where the previous CUSUM time series is either maximized or minimized (depending on the direction of the changepoint we want to locate). We continue this process until a stable changepoint is located or we exceed the maximum number of iterations.
2. Testing its statistical significance—a log-likelihood ratio test is used to test if the mean of the given time series changes at the identified changepoint. The null hypothesis states that there is no change in the mean of the series.

Some further remarks about the implementation of the algorithm:

- The algorithm can be used to detect both upward and downward shifts.
- The algorithm can find at most one upward and one downward changepoint.
- By default, the changepoints are only reported if the null hypothesis is rejected.
- Under the hood, the Gaussian distribution is used to calculate the CUSUM time series value and perform the hypothesis test.

In this recipe, we will apply the CUSUM algorithm to identify changepoints in Apple's stock prices from 2020.

How to do it...

Execute the following steps to detect changepoints in Apple's stock price:

1. Import the libraries:

```
import yfinance as yf
from kats.detectors.cusum_detection import CUSUMDetector
from kats.consts import TimeSeriesData
```

2. Download Apple's stock price from 2020:

```
df = yf.download("AAPL",
                 start="2020-01-01",
                 end="2020-12-31",
                 progress=False)
```

3. Keep only the adjusted close price, reset the index, and rename the columns:

```
df = df[["Adj Close"]].reset_index(drop=False)
df.columns = ["time", "price"]
```

4. Convert the DataFrame into a TimeSeriesData object:

```
tsd = TimeSeriesData(df)
```

5. Instantiate and run the changepoint detector:

```
cusum_detector = CUSUMDetector(tsd)
change_points = cusum_detector.detector(
    change_directions=["increase"]
)
cusum_detector.plot(change_points)
```

Running the code generates the following plot:

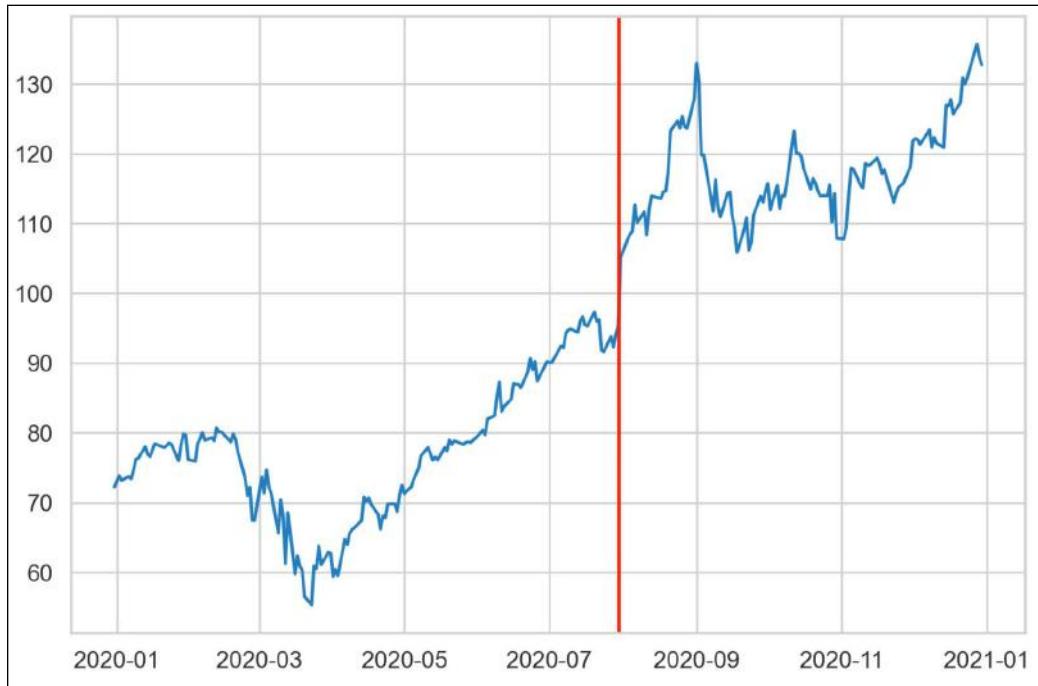


Figure 4.5: The changepoint detected by the CUSUM algorithm

We see that the algorithm picked the biggest jump as the changepoint.

6. Investigate the detected changepoint in more detail:

```
point, meta = change_points[0]  
point
```

What returns the following information about the detected changepoint:

```
TimeSeriesChangepoint(start_time: 2020-07-30 00:00:00, end_time: 2020-07-  
30 00:00:00, confidence: 1.0)
```

The identified changepoint is on the 30th of July and indeed the stock price jumped from \$95.4 on that day to \$105.4 on the next day, mostly due to a strong quarterly earnings report.

How it works...

In the first step, we imported the libraries. For detecting the changepoints, we used the kats library from Facebook. Then, we fetched Apple's stock prices from 2020. For this analysis, we used the adjusted close prices.

To work with kats, we need to have our data in a particular format. That is why in *Step 3*, we only kept the adjusted close prices, reset the index without dropping it (as we need that column), and renamed the columns. One thing to remember is that the column containing the dates/datetimes must be called `time`. In *Step 4*, we converted the DataFrame into a `TimeSeriesData` object, which is a representation used by kats.

In *Step 5*, we instantiated the `CUSUMDetector` using the previously created data. We did not change any default settings. Then, we identified the changepoints using the `detector` method. For this analysis, we were only interested in increases, so we specified the `change_directions` argument. Lastly, we plotted the detected changepoint using the `plot` method of the `cusum_detector` object. One thing to note here is that we had to provide the identified changepoints as input for the method.

In the very last step, we looked further into the detected changepoints. The returned object is a list containing two elements: the `TimeSeriesChangePoint` object containing information, such as the date of the identified changepoint and the algorithm's confidence, and a metadata object. By using the latter's `__dict__` method, we can access more information about the point: the direction, the mean before/after the changepoint, the p-value of the likelihood ratio test, and more.

There's more...

The library offers quite a few more interesting functionalities regarding changepoint detection. We will only cover two of them, and I highly encourage you to explore them further on your own.

Restricting the detection window

The first one is to restrict the window in which we want to look for the changepoint. We can do so using the `interest_window` argument of the `detector` method. Below, we only looked for a changepoint between the 200th and 250th observations (reminder: this is a trading year and not a full calendar year, so there are only around 252 observations).

Narrow down the window in which we want to search for the changepoint:

```
change_points = cusum_detector.detector(change_directions=["increase"],
                                         interest_window=[200, 250])
cusum_detector.plot(change_points)
```

We can see the modified results in the following plot:

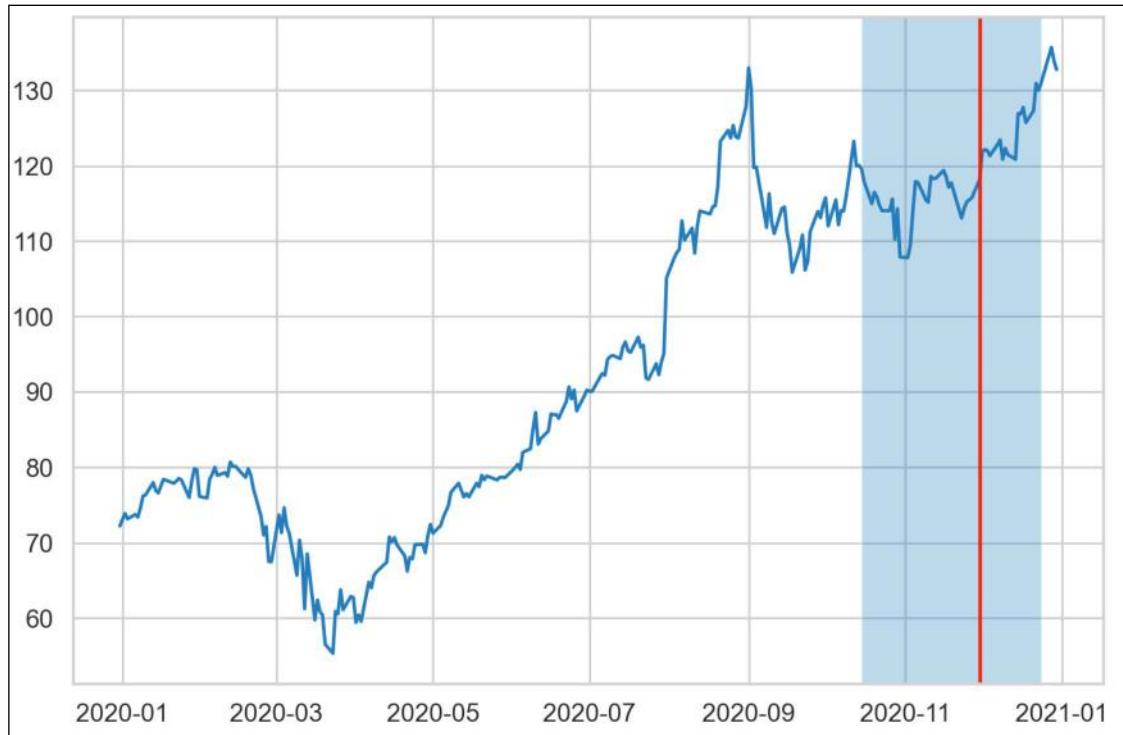


Figure 4.6: Changepoint identified between the 200th and 250th observations in the series

Aside from the identified changepoint, we can also see the window we have selected.

Using different changepoint detection algorithms

The kats library also contains other interesting algorithms for changepoint detection. One of them is `RobustStatDetector`. Without going into too much detail about the algorithm itself, it smoothens the data using moving averages before identifying the points of interest. Another interesting feature of the algorithm is that it can detect multiple changepoints in a single run.

Use another algorithm to detect changepoints (`RobustStatDetector`):

```
from kats.detectors.robust_stat_detection import RobustStatDetector

robust_detector = RobustStatDetector(tsd)
change_points = robust_detector.detector()
robust_detector.plot(change_points)
```

Running the snippet generates the following plot:

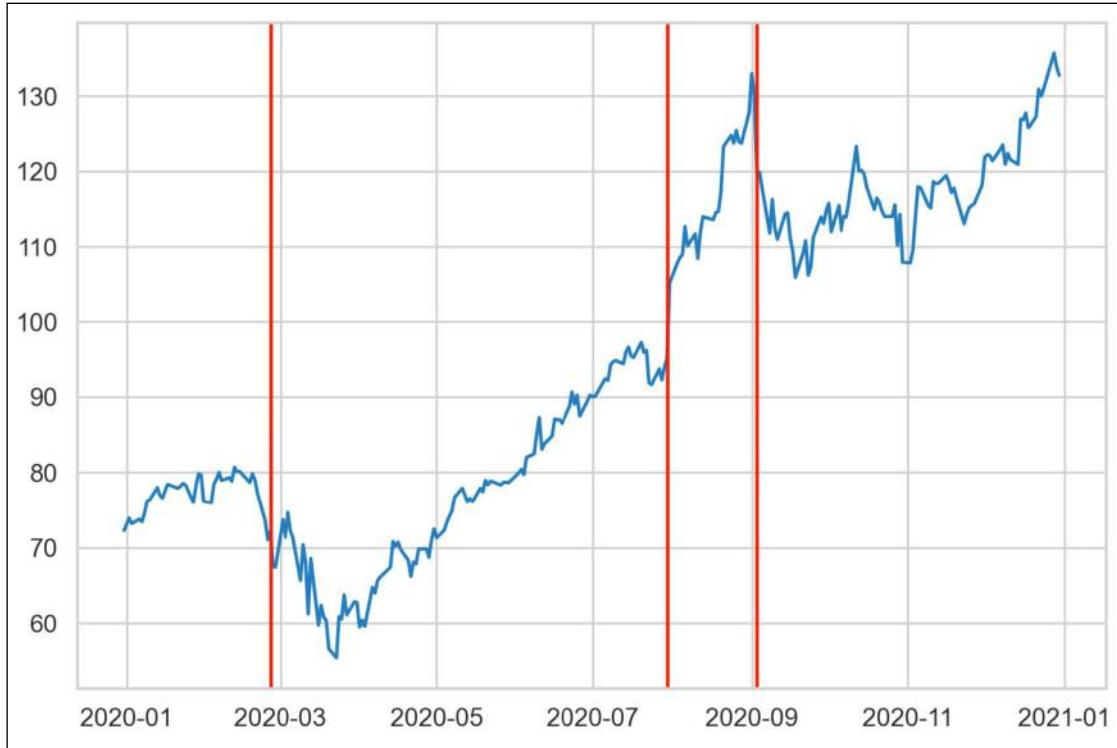


Figure 4.7: Identifying changepoints using the RobustStatDetector

This time, the algorithm picked up two additional changepoints compared to the previous attempt.



Another interesting algorithm provided by the `kats` library is the **Bayesian Online Change Point Detection (BOCPD)**, for which we provide a reference in the *See also* section.

See also

- <https://github.com/facebookresearch/Kats>—the GitHub repository of Facebook’s Kats
- Page, E. S. 1954. “Continuous inspection schemes.” *Biometrika* 41(1): 100–115
- Adams, R. P., & MacKay, D. J. (2007). *Bayesian online changepoint detection*. arXiv preprint arXiv:0710.3742

Detecting trends in time series

In the previous recipe, we covered changepoint detection. Another class of algorithms can be used for trend detection, that is, identifying significant and prolonged changes in time series.

The `kats` library offers a trend detection algorithm based on the non-parametric **Mann-Kendall (MK)** test. The algorithm iteratively conducts the MK test on windows of a specified size and returns the starting points of each window for which this test turned out to be statistically significant.

To detect whether there is a significant trend in the window, the test inspects the monotonicity of the increases/decreases in the time series rather than the magnitude of the change in values. The MK test uses a test statistic called Kendall's Tau, and it ranges from -1 to 1. We can interpret the values as follows:

- -1 indicates a perfectly monotonic decline
- 1 indicates a perfectly monotonic increase
- 0 indicates that there is no directional trend in the series

By default, the algorithm will only return periods in which the results were statistically significant.

You might be wondering why use an algorithm for detecting trends when it is easy to see them on the plot? That is very true; however, we should remember that the goal of using those algorithms is to look at more than a single series and time period at a time. We want to be able to detect trends at scale, for example, finding upward trends within hundreds of time series.

In this recipe, we will use the trend detection algorithm to investigate whether there were periods with significant increasing trends in NVIDIA's stock prices from 2020.

How to do it...

Execute the following steps to detect increasing trends in NVIDIA's stock prices from 2020:

1. Import the libraries:

```
import yfinance as yf
from kats.consts import TimeSeriesData
from kats.detectors.trend_mk import MKDetector
```

2. Download NVIDIA's stock prices from 2020:

```
df = yf.download("NVDA",
                 start="2020-01-01",
                 end="2020-12-31",
                 progress=False)
```

3. Keep only the adjusted close price, reset the index, and rename the columns:

```
df = df[["Adj Close"]].reset_index(drop=False)
df.columns = ["time", "price"]
```

4. Convert the DataFrame into a TimeSeriesData object:

```
tsd = TimeSeriesData(df)
```

5. Instantiate and run the trend detector:

```
trend_detector = MKDetector(tsd, threshold=0.9)
time_points = trend_detector.detector(
    direction="up",
    window_size=30
)
```

6. Plot the detected time points:

```
trend_detector.plot(time_points)
```

Running the line results in the following plot:

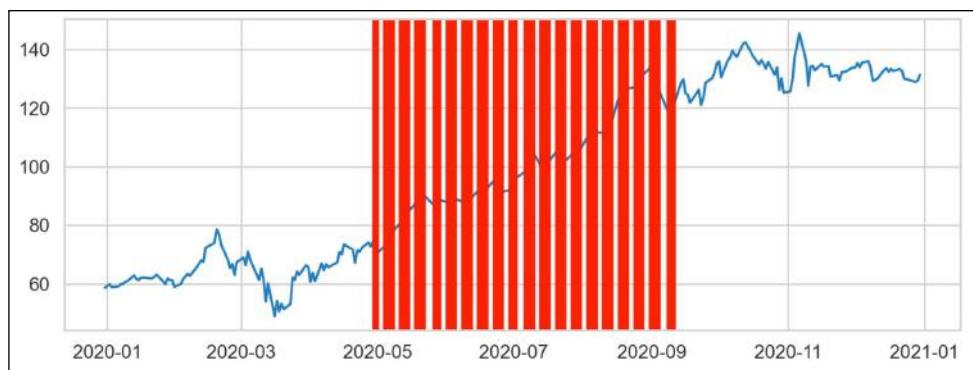


Figure 4.8: The identified starting points of upward trends

In Figure 4.8, we can see a lot of periods with some gaps in between. What is important to know is that the red vertical bars are not the detected windows, but rather a lot of detected trend starting points right next to each other. Running the selected configuration of the algorithm on our data resulted in identifying 95 periods of an increasing trend, which clearly have a lot of overlap.

How it works...

The first four steps are very similar to the previous recipe, with the only difference being that this time we downloaded NVIDIA's stock price from 2020. Please refer to the previous recipe for more information about preparing the data for working with the kats library.

In Step 5, we instantiated the trend detector (`MKDetector` class) while providing the data and changing the threshold of the Tau coefficient to 0.9. This way, we obtain only the periods with higher trend intensity. Then, we used the `detector` method to find the time points. We were interested in increasing trends (`direction="up"`) over a window of 30 days.



There are also other parameters of the detector we could tune. For example, we can specify if there is some seasonality in the data by using the `freq` argument.

In *Step 6*, we plotted the results. We can also inspect in detail each of the 95 detected points. The returned `time_points` object is a list of tuples, in which each tuple contains the `TimeSeriesChangePoint` object (with the beginning date of the detected trend period) and the point's metadata. In our case, we looked for periods of an increasing trend over a window of 30 days. Naturally, there will be quite an overlap in the periods of an increasing trend, as we identified multiple points, with each being the beginning of the period. As we can see in the plot, a lot of those identified points are consecutive.

See also

- Mann, H. B. 1945. "Non-Parametric Tests against Trend." *Econometrica* 13: 245-259.
- Kendall, M. G. 1948. *Rank Correlation Methods*. Griffin.

Detecting patterns in a time series using the Hurst exponent

In finance, a lot of trading strategies are based on one of the following:

- **Momentum**—the investors try to use the continuance of the existing market trend to determine their positions
- **Mean-reversion** – the investors assume that properties such as stock returns and volatility will revert to their long-term average over time (also known as an Ornstein-Uhlenbeck process)

While we can relatively easily classify a time series as one of the two by inspecting it visually, this solution definitely does not scale well. That is why we can use approaches such as the Hurst exponent to identify if a given time series (not necessarily a financial one) is trending, mean-reverting, or simply a random walk.



A random walk is a process in which a path consists of a succession of steps taken at random. Applied to stock prices, it suggests that changes in stock prices have the same distribution and are independent of each other. This implies that the past movement (or trend) of a stock price cannot be used to predict its future movement. For more information, please see *Chapter 10, Monte Carlo Simulations in Finance*.

Hurst exponent (H) is a measure for the long-term memory of a time series, that is, it measures the amount by which that series deviates from a random walk. The values of the Hurst exponent range between 0 and 1, with the following interpretation:

- $H < 0.5$ —a series is mean-reverting. The closer the value is to 0, the stronger the mean-reversion process is.
- $H = 0.5$ —a series is a geometric random walk.

- $H > 0.5$ —a series is trending. The closer the value is to 1, the stronger the trend.

There are a few ways of calculating the Hurst exponent. In this recipe, we will focus on the one based on estimating the rate of the diffusive behavior, which is based on the variance of log prices. For the practical example, we will use 20 years of daily S&P 500 prices.

How to do it...

Execute the following steps to investigate whether S&P 500 prices are trending, mean-reverting, or an example of a random walk:

1. Import the libraries:

```
import yfinance as yf  
import numpy as np  
import pandas as pd
```

2. Download S&P 500's historical prices from the years 2000 to 2019:

```
df = yf.download("^GSPC",  
                 start="2000-01-01",  
                 end="2019-12-31",  
                 progress=False)  
df["Adj Close"].plot(title="S&P 500 (years 2000-2019)")
```

Running the code generates the following plot:

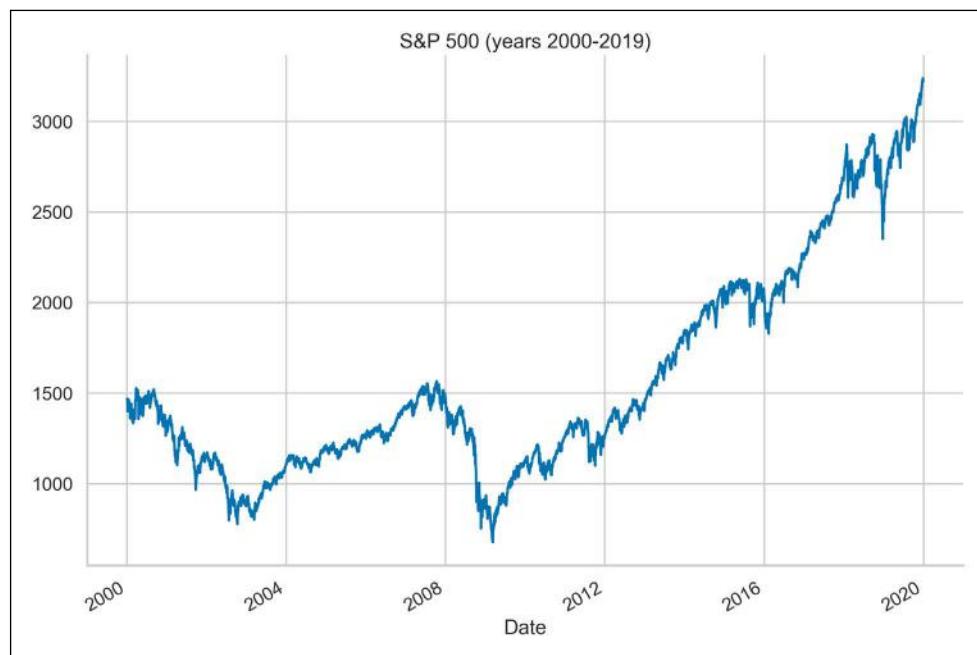


Figure 4.9: The S&P 500 index in the years 2000 to 2019

We plot the data to get some initial intuition of what to expect from the calculated Hurst exponent.

3. Define a function calculating the Hurst exponent:

```
def get_hurst_exponent(ts, max_lag=20):
    """Returns the Hurst Exponent of the time series"""
    lags = range(2, max_lag)
    tau = [np.std(np.subtract(ts[lag:], ts[:-lag])) for lag in lags]
    hurst_exp = np.polyfit(np.log(lags), np.log(tau), 1)[0]

    return hurst_exp
```

4. Calculate the values of the Hurst exponent using different values for the `max_lag` parameter:

```
for lag in [20, 100, 250, 500, 1000]:
    hurst_exp = get_hurst_exponent(df["Adj Close"].values, lag)
    print(f"Hurst exponent with {lag} lags: {hurst_exp:.4f}")
```

This returns the following:

```
Hurst exponent with 20 lags: 0.4478
Hurst exponent with 100 lags: 0.4512
Hurst exponent with 250 lags: 0.4917
Hurst exponent with 500 lags: 0.5265
Hurst exponent with 1000 lags: 0.5180
```

The more lags we include, the closer we get to the verdict that the S&P 500 series is a random walk.

5. Narrow down the data to the years 2005 to 2007 and calculate the exponents one more time:

```
shorter_series = df.loc["2005":"2007", "Adj Close"].values
for lag in [20, 100, 250, 500]:
    hurst_exp = get_hurst_exponent(shorter_series, lag)
    print(f"Hurst exponent with {lag} lags: {hurst_exp:.4f}")
```

This returns the following:

```
Hurst exponent with 20 lags: 0.3989
Hurst exponent with 100 lags: 0.3215
Hurst exponent with 250 lags: 0.2507
Hurst exponent with 500 lags: 0.1258
```

It seems that the series from the 2005 to 2007 period is mean-reverting. For reference, the discussed time series is illustrated as follows:

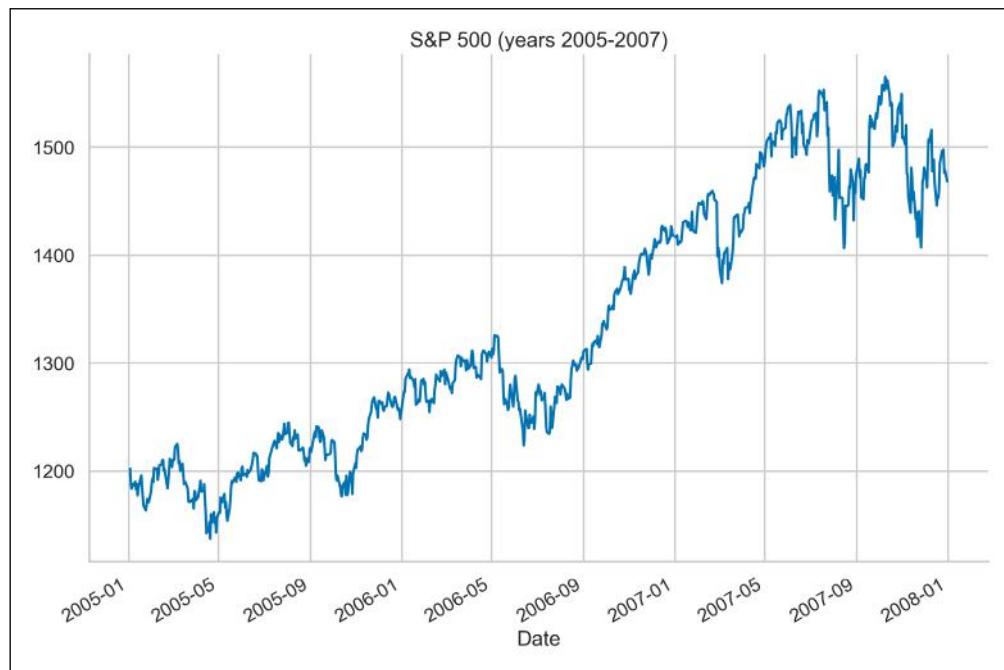


Figure 4.10: The S&P 500 index in the years 2005 to 2007

How it works...

After importing the required libraries, we downloaded 20 years' worth of daily S&P prices from Yahoo Finance. From looking at the plot, it is hard to say whether the time series is purely trending, mean-reverting, or a random walk. There seems to be a clear rising trend, especially in the second half of the series.

In *Step 3*, we defined a function used for calculating the Hurst exponent. For this approach, we need to provide the maximum number of lags to be used for the calculations. This parameter greatly impacts the results, as we will see later on.

The calculations of the Hurst exponent can be summarized in two steps:

1. For each lag in the considered range, we calculate the standard deviation of the differenced series (we will cover differencing in more depth in *Chapter 6, Time Series Analysis and Forecasting*).
2. Calculate the slope of the log plot of lags versus the standard deviations to get the Hurst exponent.

In *Step 4*, we calculated and printed the Hurst exponent for a range of different values of the `max_lag` parameter. For lower values of the parameter, the series could be considered slightly mean-reverting. While increasing the value of the parameter, the interpretation changed more in favor of the series being a random walk.

In *Step 5*, we carried out a similar experiment, but this time on a restricted time series. We only looked at data from the years 2005 to 2007. We also had to remove the `max_lag` of 1,000, given there were not enough observations in the restricted time series. As we could see, the results have changed a bit more drastically than before, from 0.4 for `max_lag` of 20 to 0.13 for 500 lags.

While using the Hurst exponent for our analyses, we should keep in mind that the results can vary depending on:

- The method we use for calculating the Hurst exponent
- The value of the `max_lag` parameter
- The period we are looking at – local patterns can be very different from the global ones

There's more...

As we mentioned in the introduction, there are multiple ways to calculate the Hurst exponent. Another quite popular approach is to use the **rescaled range (R/S) analysis**. A brief literature review suggests that using the R/S statistic leads to better results in comparison to other methods such as the analysis of autocorrelations, variance ratios, and more. A possible shortcoming of that method is that it is very sensitive to short-range dependencies.

For an implementation of the Hurst exponent based on the rescaled range analysis, you can check out the `hurst` library.

See also

- <https://github.com/Mottl/hurst>—the repository of the `hurst` library
- Hurst, H. E. 1951. “Long-Term Storage Capacity of Reservoirs.” *ASCE Transactions* 116(1): 770–808
- Kroha, P., & Skoula, M. 2018, March. *Hurst Exponent and Trading Signals Derived from Market Time Series*. In *ICEIS* (1): 371–378

Investigating stylized facts of asset returns

Stylized facts are statistical properties that are present in many empirical asset returns (across time and markets). It is important to be aware of them because when we are building models that are supposed to represent asset price dynamics, the models should be able to capture/replicate these properties.

In this recipe, we investigate the five stylized facts using an example of daily S&P 500 returns from the years 2000 to 2020.

Getting ready

As this is a longer recipe with further subsections, we import the required libraries and prepare the data in this section:

1. Import the required libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
import seaborn as sns
import scipy.stats as scs
import statsmodels.api as sm
import statsmodels.tsa.api as smt
```

2. Download the S&P 500 data and calculate the returns:

```
df = yf.download("^GSPC",
                 start="2000-01-01",
                 end="2020-12-31",
                 progress=False)

df = df[["Adj Close"]].rename(
    columns={"Adj Close": "adj_close"}
)
df["log rtn"] = np.log(df["adj_close"]/df["adj_close"].shift(1))
df = df[["adj_close", "log rtn"]].dropna()
```

How to do it...

In this section, we sequentially investigate the five stylized facts in the S&P 500 returns series:

Fact 1: Non-Gaussian distribution of returns

It was observed in the literature that (daily) asset returns exhibit the following:

- **Negative skewness (third moment):** Large negative returns occur more frequently than large positive ones
- **Excess kurtosis (fourth moment):** Large (and small) returns occur more often than expected under normality



Moments are a set of statistical measures used to describe a probability distribution. The first four moments are the following: expected value (mean), variance, skewness, and kurtosis.

Run the following steps to investigate the existence of the first fact by plotting the histogram of returns and a **quantile-quantile** (Q-Q) plot:

1. Calculate the Normal probability density function (**PDF**) using the mean and standard deviation of the observed returns:

```
r_range = np.linspace(min(df["log rtn"]),
                      max(df["log rtn"]),
                      num=1000)
mu = df["log rtn"].mean()
sigma = df["log rtn"].std()
norm_pdf = scs.norm.pdf(r_range, loc=mu, scale=sigma)
```

2. Plot the histogram and the Q-Q plot:

```
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

# histogram
sns.distplot(df.log_rtn, kde=False,
              norm_hist=True, ax=ax[0])
ax[0].set_title("Distribution of S&P 500 returns",
                 fontsize=16)
ax[0].plot(r_range, norm_pdf, "g", lw=2,
            label=f"N({mu:.2f}, {sigma**2:.4f})")
ax[0].legend(loc="upper left")

# Q-Q plot
qq = sm.qqplot(df.log_rtn.values, line="s", ax=ax[1])
ax[1].set_title("Q-Q plot", fontsize=16)

plt.show()
```

Executing the code results in the following plot:

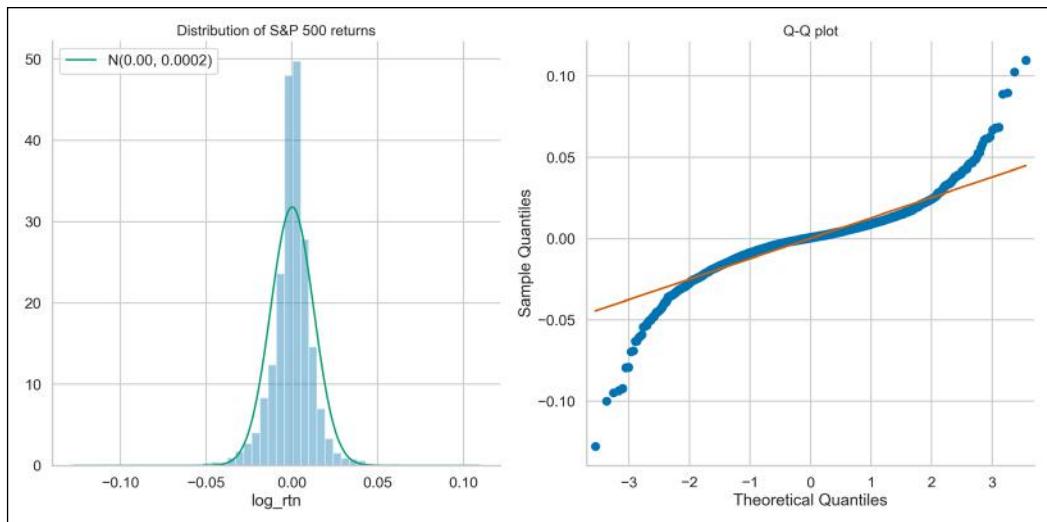


Figure 4.11: The distribution of S&P 500 returns visualized using a histogram and a Q-Q plot

We can use the histogram (showing the shape of the distribution) and the Q-Q plot to assess the normality of the log returns. Additionally, we can print the summary statistics (please refer to the GitHub repository for the code):

```
----- Descriptive Statistics -----
Range of dates: 2000-01-03 - 2020-12-30
Number of observations: 5283
Mean: 0.0002
Median: 0.0006
Min: -0.1277
Max: 0.1096
Standard Deviation: 0.0126
Skewness: -0.3931
Kurtosis: 10.9531
Jarque-Bera statistic: 26489.07 with p-value: 0.00
```

By looking at the metrics such as the mean, standard deviation, skewness, and kurtosis, we can infer that they deviate from what we would expect under normality. The four moments of the Standard Normal distribution are 0, 1, 0, and 0 respectively. Additionally, the Jarque-Bera normality test gives us reason to reject the null hypothesis by stating that the distribution is normal at the 99% confidence level.

The fact that the returns do not follow the Normal distribution is crucial, given many statistical models and approaches assume that the random variable is normally distributed.

Fact 2: Volatility clustering

Volatility clustering is the pattern in which large changes in prices tend to be followed by large changes (periods of higher volatility), while small changes in price are followed by small changes (periods of lower volatility).

Run the following code to investigate the second fact by plotting the log returns series:

```
(  
    df["log_rtn"]  
    .plot(title="Daily S&P 500 returns", figsize=(10, 6))  
)
```

Executing the code results in the following plot:

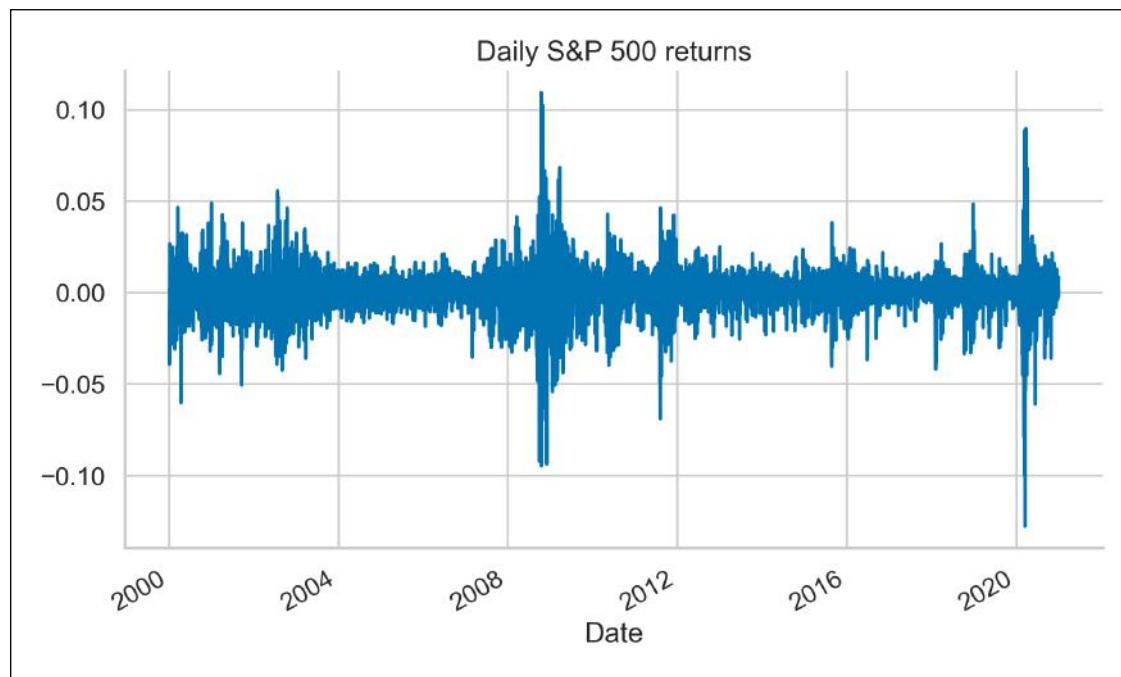


Figure 4.12: Examples of volatility clustering in the S&P 500 returns

We can observe clear clusters of volatility—periods of higher positive and negative returns. The fact that volatility is not constant and that there are some patterns in how it evolves is a very useful observation when we attempt to forecast volatility, for example, using GARCH models. For more information, please refer to *Chapter 9, Modeling Volatility with GARCH Class Models*.

Fact 3: Absence of autocorrelation in returns

Autocorrelation (also known as serial correlation) measures how similar a given time series is to the lagged version of itself over successive time intervals.

Below, we investigate the third fact by stating the absence of autocorrelation in returns:

1. Define the parameters for creating the autocorrelation plots:

```
N_LAGS = 50  
SIGNIFICANCE_LEVEL = 0.05
```

2. Run the following code to create the **autocorrelation function (ACF)** plot of log returns:

```
acf = smt.graphics.plot_acf(df["log rtn"],  
                           lags=N_LAGS,  
                           alpha=SIGNIFICANCE_LEVEL)  
plt.show()
```

Executing the snippet results in the following plot:

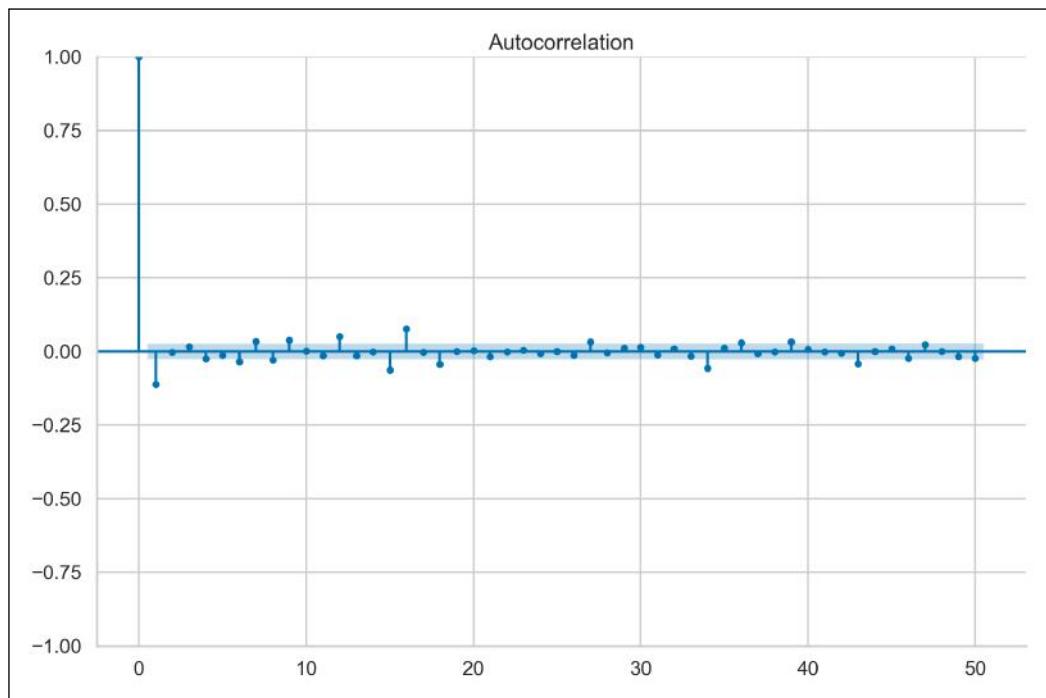


Figure 4.13: The plot of the autocorrelation function of the S&P 500 returns

Only a few values lie outside of the confidence interval (we do not look at lag 0) and can be considered statistically significant. We can assume that we have verified that there is no autocorrelation in the log-returns series.

Fact 4: Small and decreasing autocorrelation in squared/absolute returns

While we expect no autocorrelation in the return series, it was empirically proven that we can observe small and slowly decaying autocorrelation (also referred to as persistence) in simple nonlinear functions of the returns, such as absolute or squared returns. This observation is connected to the phenomenon we have already investigated, that is, volatility clustering.

The autocorrelation function of the squared returns is a common measure of volatility clustering. It is also referred to as the ARCH effect, as it is the key component of (G)ARCH models, which we cover in *Chapter 9, Modeling Volatility with GARCH Class Models*. However, we should keep in mind that this property is model-free and not exclusively connected to GARCH class models.

We can investigate the fourth fact by creating the ACF plots of squared and absolute returns:

```
fig, ax = plt.subplots(2, 1, figsize=(12, 10))

smt.graphics.plot_acf(df["log rtn"]**2, lags=N_LAGS,
                      alpha=SIGNIFICANCE_LEVEL, ax=ax[0])
ax[0].set(title="Autocorrelation Plots",
           ylabel="Squared Returns")

smt.graphics.plot_acf(np.abs(df["log rtn"]), lags=N_LAGS,
                      alpha=SIGNIFICANCE_LEVEL, ax=ax[1])
ax[1].set(ylabel="Absolute Returns",
           xlabel="Lag")

plt.show()
```

Executing the code results in the following plots:

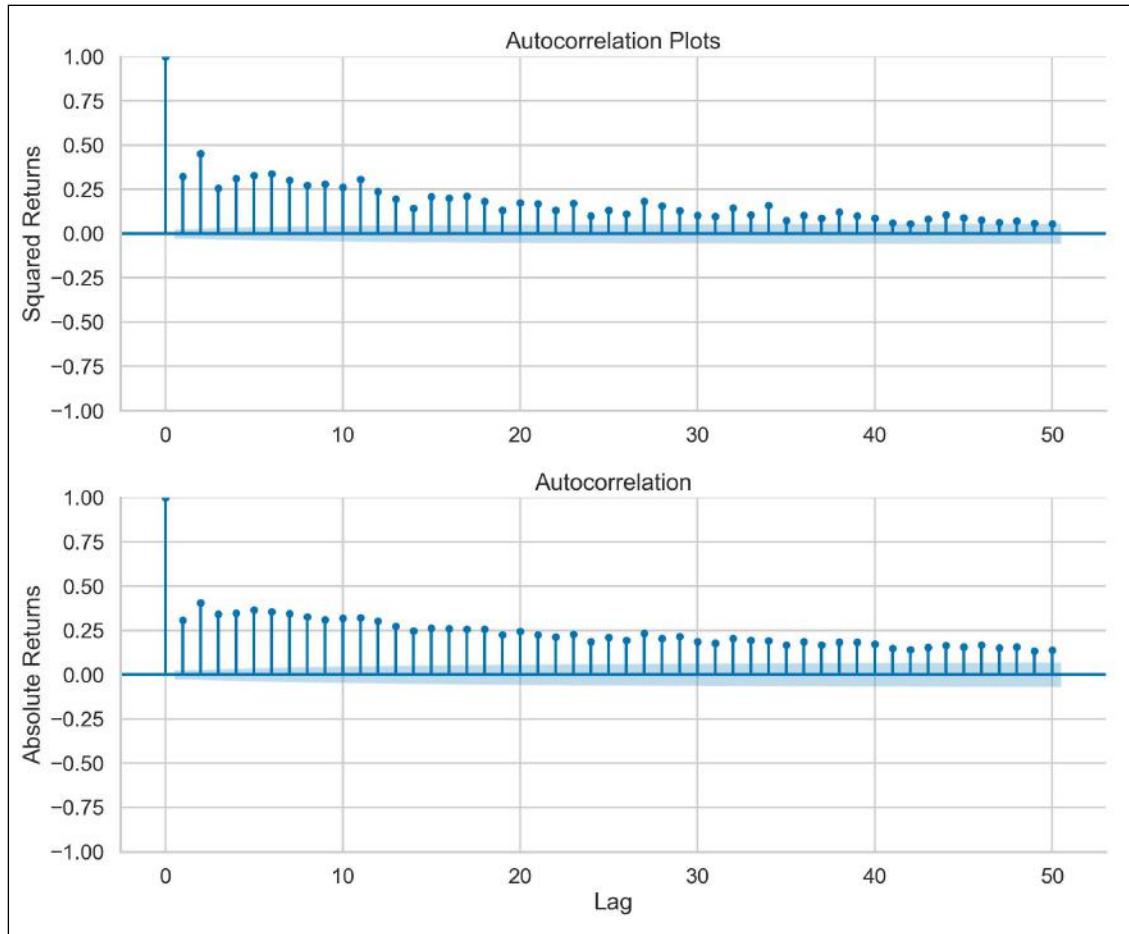


Figure 4.14: The ACF plots of squared and absolute returns

We can observe the small and decreasing values of autocorrelation for the squared and absolute returns, which are in line with the fourth stylized fact.

Fact 5: Leverage effect

The **leverage effect** refers to the fact that most measures of an asset's volatility are negatively correlated with its returns.

Execute the following steps to investigate the existence of the leverage effect in the S&P 500's return series:

1. Calculate volatility measures as moving standard deviations:

```
df["moving_std_252"] = df[["log rtn"]].rolling(window=252).std()  
df["moving_std_21"] = df[["log rtn"]].rolling(window=21).std()
```

2. Plot all the series:

```
fig, ax = plt.subplots(3, 1, figsize=(18, 15),  
                      sharex=True)  
  
df["adj_close"].plot(ax=ax[0])  
ax[0].set(title="S&P 500 time series",  
          ylabel="Price ($)")  
  
df["log rtn"].plot(ax=ax[1])  
ax[1].set(ylabel="Log returns")  
  
df["rolling_std_252"].plot(ax=ax[2], color="r",  
                           label="Rolling Volatility 252d")  
df["rolling_std_21"].plot(ax=ax[2], color="g",  
                           label="Rolling Volatility 21d")  
ax[2].set(ylabel="Moving Volatility",  
          xlabel="Date")  
ax[2].legend()  
  
plt.show()
```

We can now investigate the leverage effect by visually comparing the price series to the (rolling) volatility metric:

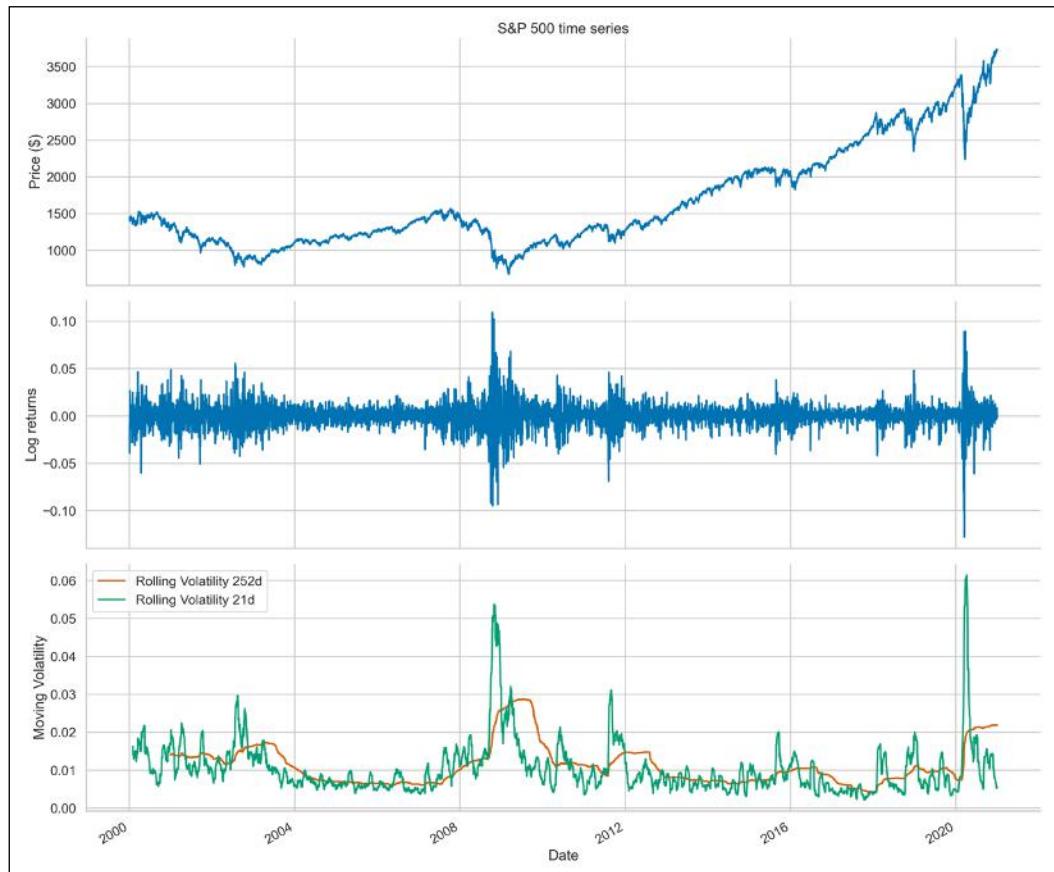


Figure 4.15: The rolling volatility metrics of the S&P 500 returns

In *Figure 4.15*, we can observe a pattern of increased volatility when the prices go down and decreased volatility when they are rising. This observation is in line with the fact's definition.

How it works...

In this section, we describe the approaches we used to investigate the existence of the stylized facts in the S&P 500 log return series.

Fact 1: Non-Gaussian distribution of returns

We will break down investigating this fact into three parts.

Histogram of returns

The first step in investigating this fact was to plot a histogram by visualizing the distribution of returns. To do so, we used `sns.distplot` while setting `kde=False` (which does not use the Gaussian kernel density estimate) and `norm_hist=True` (this plot shows density instead of the count).

To see the difference between our histogram and Gaussian distribution, we superimposed a line representing the PDF of the Gaussian distribution with the mean and standard deviation coming from the considered return series.

First, we specified the range over which we calculated the PDF by using `np.linspace` (we set the number of points to 1,000; generally, the more points, the smoother the line) and then calculated the PDF using the `scs.norm.pdf` function. The default arguments correspond to the standard normal distribution, that is, with zero mean and unit variance. That is why we specified the `loc` and `scale` arguments as the sample mean and standard deviation, respectively.

To verify the existence of the previously mentioned patterns, we should look at the following:

- Negative skewness: The left tail of the distribution is longer, while the mass of the distribution is concentrated on the right-hand side of the distribution
- Excess kurtosis: Fat-tailed and peaked distribution

The second point is easier to observe on our plot, as there is a clear peak over the PDF, and we see more mass in the tails.

Q-Q plot

After inspecting the histogram, we looked at the Q-Q plot, on which we compared two distributions (theoretical and observed) by plotting their quantiles against each other. In our case, the theoretical distribution is Gaussian (normal), and the observed one comes from the S&P 500 returns.

To obtain the plot, we used the `sm.qqplot` function. If the empirical distribution is normal, then the vast majority of the points will lie on the red line. However, we see that this is not the case, as points on the left-hand side of the plot are more negative (that is, lower empirical quantiles are smaller) than expected in the case of the Gaussian distribution (indicated by the line). This means that the left tail of the returns distribution is heavier than that of the Gaussian distribution. Analogical conclusions can be drawn about the right tail, which is heavier than under normality.

Descriptive statistics

The last part involves looking at some statistics. We calculated them using the appropriate methods of pandas Series/DataFrames. We immediately saw that the returns exhibit negative skewness and excess kurtosis. We also ran the Jarque-Bera test (`scs.jarque_bera`) to verify that the returns do not follow a Gaussian distribution. With a p-value of zero, we rejected the null hypothesis that sample data has skewness and kurtosis matching those of a Gaussian distribution.



The pandas implementation of kurtosis is the one that literature refers to as excess kurtosis or Fisher's kurtosis. Using this metric, the excess kurtosis of a Gaussian distribution is 0, while the standard kurtosis is 3. This is not to be confused with the name of the stylized fact's excess kurtosis, which simply means kurtosis higher than that of normal distribution.

Fact 2: Volatility clustering

Another thing we should be aware of when investigating stylized facts is volatility clustering—periods of high returns alternating with periods of low returns, suggesting that volatility is not constant. To quickly investigate this fact, we plot the returns using the `plot` method of a pandas DataFrame.

Fact 3: Absence of autocorrelation in returns

To investigate whether there is significant autocorrelation in returns, we created the autocorrelation plot using `plot_acf` from the `statsmodels` library. We inspected 50 lags and used the default `alpha=0.05`, which means that we also plotted the 95% confidence interval. Values outside of this interval can be considered statistically significant.

Fact 4: Small and decreasing autocorrelation in squared/absolute returns

To verify this fact, we also used the `plot_acf` function from the `statsmodels` library. However, this time, we applied it to the squared and absolute returns.

Fact 5: Leverage effect

This fact states that most measures of asset volatility are negatively correlated with their returns. To investigate it, we used the moving standard deviation (calculated using the `rolling` method of a pandas DataFrame) as a measure of historical volatility. We used windows of 21 and 252 days, which correspond to one month and one year of trading data.

There's more...

We present another method of investigating the leverage effect (fact 5). To do so, we use the VIX (CBOE Volatility Index), which is a popular metric of the stock market's expectations regarding volatility. The measure is implied by option prices on the S&P 500 index. We take the following steps:

1. Download and preprocess the prices of the S&P 500 and VIX:

```
df = yf.download(["^GSPC", "^VIX"],
                 start="2000-01-01",
                 end="2020-12-31",
                 progress=False)

df = df[["Adj Close"]]
df.columns = df.columns.droplevel(0)
df = df.rename(columns={"^GSPC": "sp500", "^VIX": "vix"})
```

2. Calculate the log returns (we can just as well use simple returns):

```
df["log rtn"] = np.log(df["sp500"] / df["sp500"].shift(1))
df["vol rtn"] = np.log(df["vix"] / df["vix"].shift(1))
df.dropna(how="any", axis=0, inplace=True)
```

3. Plot a scatterplot with the returns on the axes and fit a regression line to identify the trend:

```
corr_coeff = df.log_rtn.corr(df.vol_rtn)

ax = sns.regplot(x="log_rtn", y="vol_rtn", data=df,
                  line_kws={"color": "red"})
ax.set(title=f"S&P 500 vs. VIX ($\rho$ = {corr_coeff:.2f})",
       ylabel="VIX log returns",
       xlabel="S&P 500 log returns")

plt.show()
```

We additionally calculated the correlation coefficient between the two series and included it in the title:

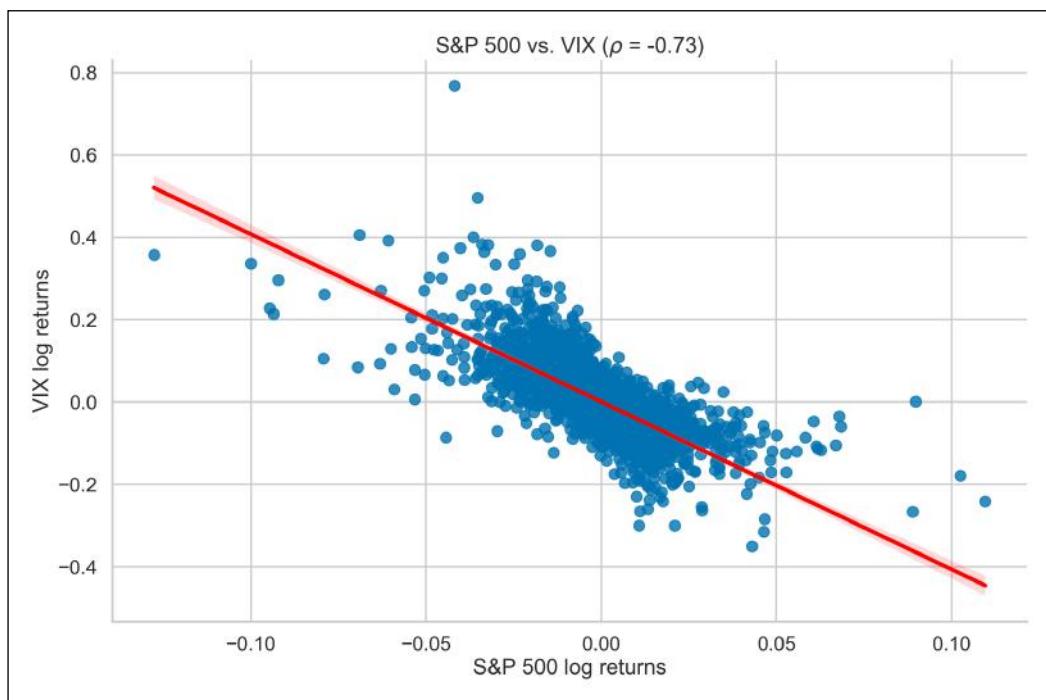


Figure 4.16: Investigating the relationship between the returns of S&P 500 and VIX

We can see that both the negative slope of the regression line and a strong negative correlation between the two series confirm the existence of the leverage effect in the return series.

See also

For more information, refer to the following:

- Cont, R. 2001. "Empirical properties of asset returns: stylized facts and statistical issues." *Quantitative Finance*, 1(2): 223

Summary

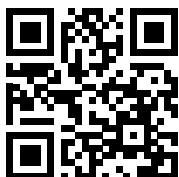
In this chapter, we learned how to use a selection of algorithms and statistical tests to automatically identify potential patterns and issues (for example, outliers) in financial time series. With their help, we can scale up our analysis to an arbitrary number of assets instead of manually inspecting each and every time series.

We also explained the stylized facts of asset returns. These are crucial to understand, as many models or strategies assume a certain distribution of the variable of interest. Most frequently, a Gaussian distribution is assumed. And as we have seen, empirical asset returns are not normally distributed. That is why we have to take certain precautions to make our analyses valid while working with such time series.

In the next chapter, we will explore the vastly popular domain of technical analysis and see what insights we can gather from analyzing the patterns in asset prices.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

5

Technical Analysis and Building Interactive Dashboards

In this chapter, we will cover the basics of **technical analysis (TA)** in Python. In short, TA is a methodology for determining (forecasting) the future direction of asset prices and identifying investment opportunities based on studying past market data (especially the prices themselves and the traded volume).

We begin by showing how to calculate some of the most popular TA indicators (with hints on how to calculate others using selected Python libraries). Additionally, we show how to download precalculated technical indicators from reliable financial data vendors. We also touch upon a subfield of TA—candlestick pattern recognition.

At the end of the chapter, we demonstrate how to create a web app, which enables us to visualize and inspect the predefined TA indicators in an interactive fashion. Then, we deploy this app to the cloud, to make it accessible for anyone from anywhere.

In this chapter, we present the following recipes:

- Calculating the most popular technical indicators
- Downloading the technical indicators
- Recognizing candlestick patterns
- Building an interactive web app for technical analysis using Streamlit
- Deploying the technical analysis app

Calculating the most popular technical indicators

There are hundreds of different technical indicators that traders use for making decisions on whether to enter or exit a position. In this recipe, we will learn how to easily calculate a few of those indicators using the `TA-Lib` library, which is the most popular library for such a task. We start with a brief introduction of a few of the selected indicators.

Bollinger bands are a statistical method, used for deriving information about the prices and volatility of a certain asset over time. To obtain the Bollinger bands, we need to calculate the moving average and standard deviation of the time series (prices), using a specified window (typically, 20 days). Then, we set the upper/lower bands at K times (typically, 2) the moving standard deviation above/below the moving average. The interpretation of the bands is quite simple: the bands widen with an increase in volatility and contract with a decrease in volatility.



The default setting of using 2 standard deviations for the bands is connected to the (empirically incorrect) assumption about the normality of returns. Under the Gaussian distribution, we would assume that when using 2 standard deviations, 95% of returns would fall within the bands.

The **relative strength index (RSI)** is an indicator that uses the closing prices of an asset to identify oversold/overbought conditions. Most commonly, the RSI is calculated using a 14-day period and is measured on a scale from 0 to 100 (it is an oscillator). Traders usually buy an asset when it is oversold (if the RSI is below 30) and sell when it is overbought (if the RSI is above 70). More extreme high/low levels, such as 80–20, are used less frequently and, at the same time, imply stronger momentum.

The last considered indicator is the **moving average convergence divergence (MACD)**. It is a momentum indicator showing the relationship between two exponential moving averages (EMA) of a given asset's price, most commonly 26- and 12-day ones. The MACD line is the difference between the fast (short period) and slow (long period) EMAs. Lastly, we calculate the MACD signal line as a 9-day EMA of the MACD line. Traders can use the crossover of the lines as a trading signal. For example, it can be considered a buy signal when the MACD line crosses the signal line from below.

Naturally, most of the indicators are not used in isolation and traders look at multiple signals before making a decision. Also, all of the indicators can be tuned further (by changing their parameters) depending on the specific goal. We will cover backtesting trading strategies based on technical indicators in another chapter.

How to do it...

Execute the following steps to calculate some of the most popular technical indicators using IBM's stock prices from 2020:

1. Import the libraries:

```
import pandas as pd  
import yfinance as yf  
import talib
```



TA-Lib is not like most Python libraries and it has a bit of a different installation process. For more information on how to do it, please refer to the GitHub repository provided in the *See also* section.

2. Download IBM's stock prices from 2020:

```
df = yf.download("IBM",
                  start="2020-01-01",
                  end="2020-12-31",
                  progress=False,
                  auto_adjust=True)
```

3. Calculate and plot the Simple Moving Average (SMA):

```
df["sma_20"] = talib.SMA(df["Close"], timeperiod=20)
(
    df[["Close", "sma_20"]]
    .plot(title="20-day Simple Moving Average (SMA)")
)
```

Running the snippet generates the following plot:

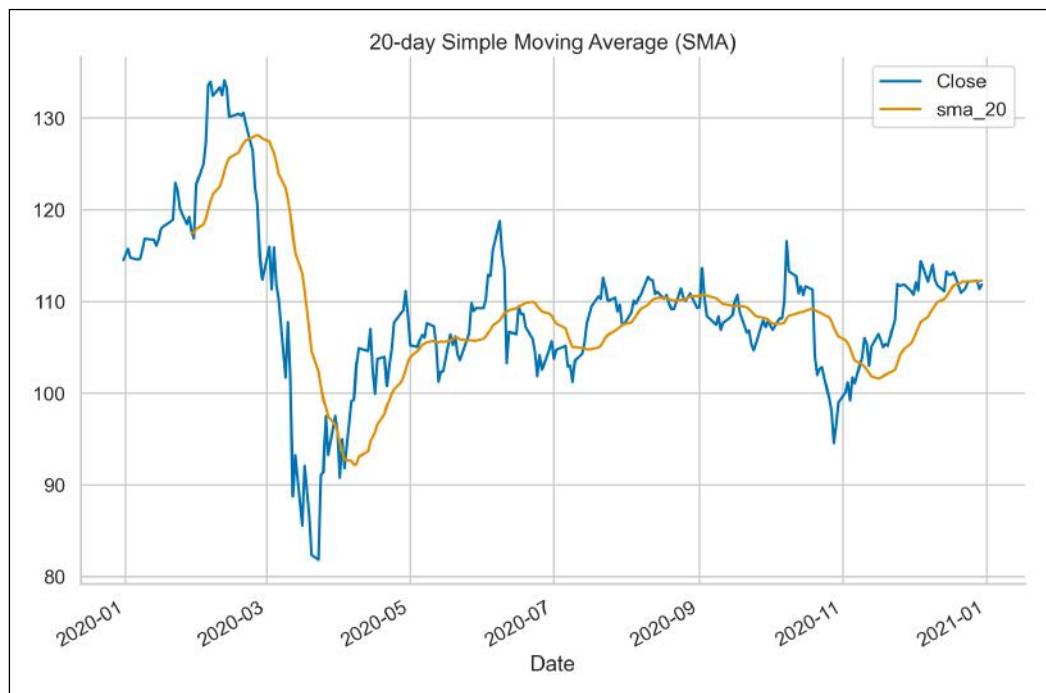


Figure 5.1: IBM's close price and the 20-day SMA

4. Calculate and plot the Bollinger bands:

```
df["bb_up"], df["bb_mid"], df["bb_low"] = talib.BBANDS(df["Close"])

fig, ax = plt.subplots()

(
    df.loc[:, ["Close", "bb_up", "bb_mid", "bb_low"]]
    .plot(ax=ax, title="Bollinger Bands")
)

ax.fill_between(df.index, df["bb_low"], df["bb_up"],
                color="gray",
                alpha=.4)
```

Running the snippet generates the following plot:

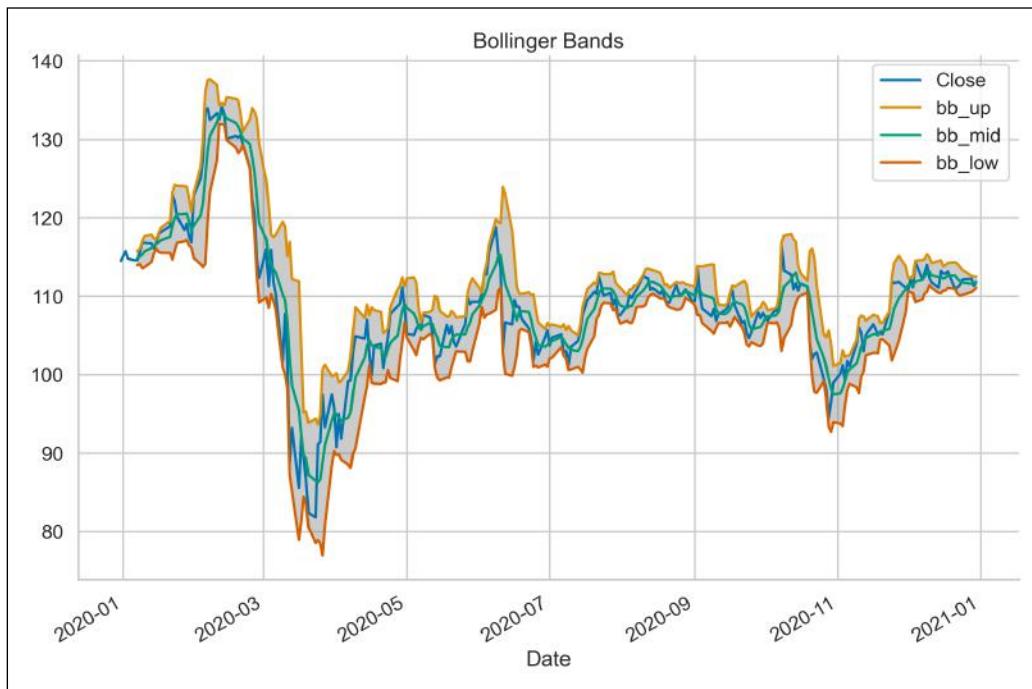


Figure 5.2: IBM's close price and the Bollinger bands

5. Calculate and plot the RSI:

```
df["rsi"] = talib.RSI(df["Close"])

fig, ax = plt.subplots()
df["rsi"].plot(ax=ax,
               title="Relative Strength Index (RSI)")
ax.hlines(y=30,
           xmin=df.index.min(),
           xmax=df.index.max(),
           color="red")
ax.hlines(y=70,
           xmin=df.index.min(),
           xmax=df.index.max(),
           color="red")
plt.show()
```

Running the snippet generates the following plot:

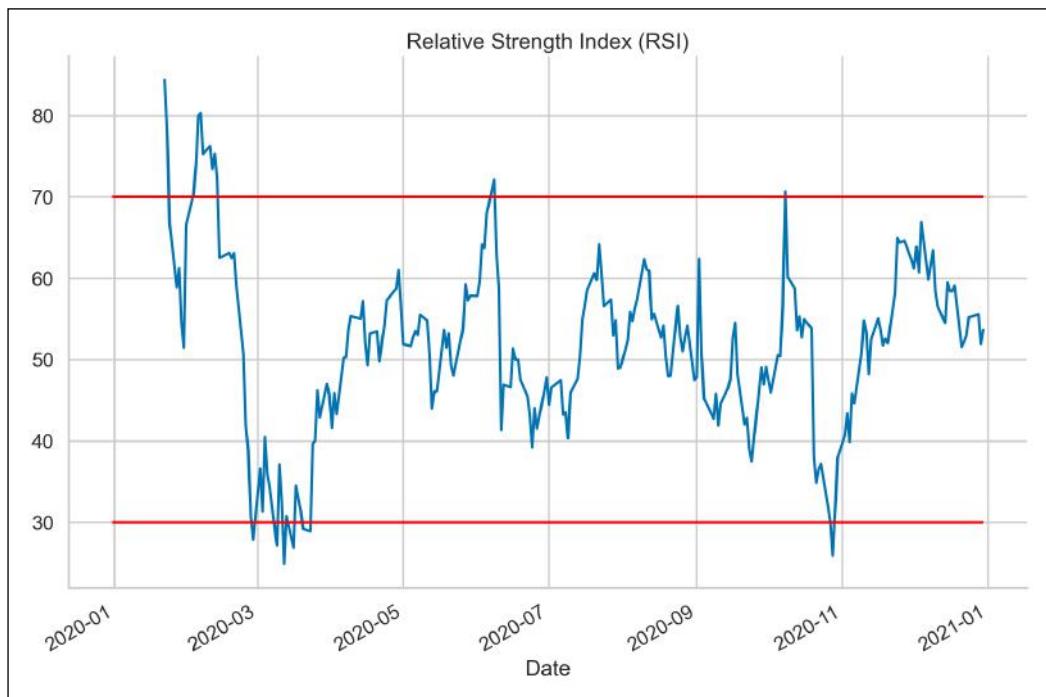


Figure 5.3: The RSI calculated using IBM's close prices

6. Calculate and plot the MACD:

```

df[ "macd"], df[ "macdsignal"], df[ "macdhist"] = talib.MACD(
    df[ "Close"], fastperiod=12, slowperiod=26, signalperiod=9
)

fig, ax = plt.subplots(2, 1, sharex=True)

(
    df[ [ "macd", "macdsignal"]].
    plot(ax=ax[0],
        title="Moving Average Convergence Divergence (MACD)")
)
ax[1].bar(df.index, df[ "macdhist"].values, label="macd_hist")
ax[1].legend()

```

Running the snippet generates the following plot:

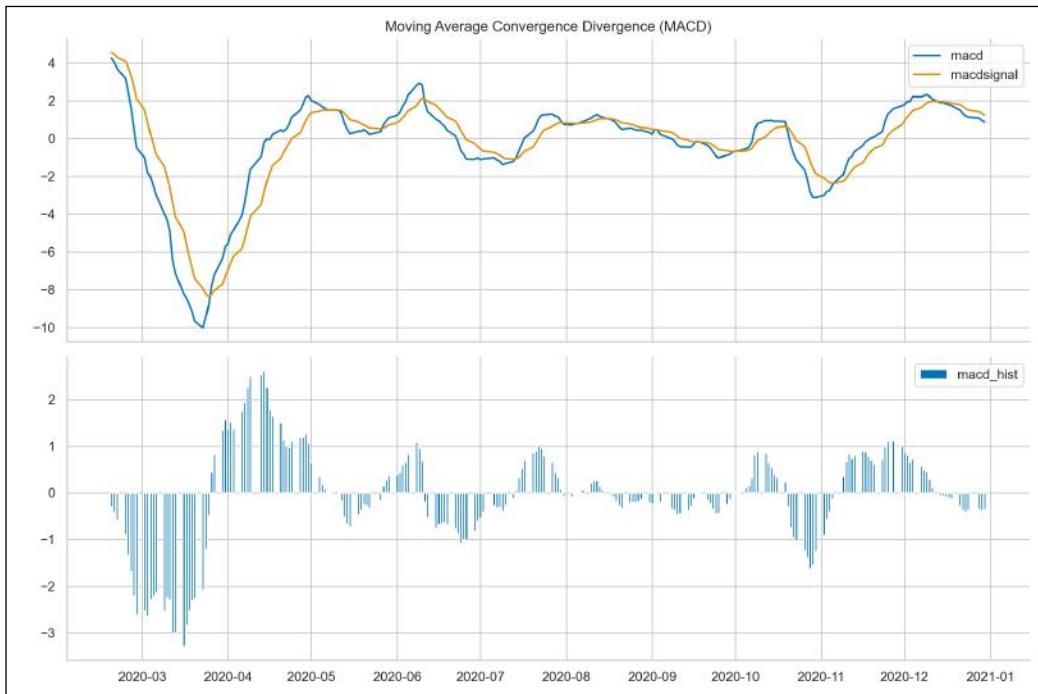


Figure 5.4: The MACD calculated using IBM's close prices

So far, we have calculated the technical indicators and plotted them. In the next chapters, we will spend more time on their implications and building trading strategies on their basis.

How it works...

After importing the libraries, we downloaded IBM's stock prices from 2020.

In *Step 3*, we calculated the 20-day simple moving average using the `SMA` function. Naturally, we could have calculated the same indicator using the `rolling` method of a `pandas` DataFrame.

In *Step 4*, we calculated the Bollinger bands. The `BBANDS` function returns three objects (the upper and lower thresholds and the moving average), which we assigned to different columns of our DataFrame.

In the next step, we calculated the RSI using the default settings. We plotted the indicator, together with two horizontal lines (created using `ax.hlines`) indicating the popular decision-making thresholds.

In the last step, we calculated the MACD, also using the default number of periods for the EMAs. The `MACD` function also returned three objects, the MACD, the signal line, and the MACD histogram, which is effectively the difference between the first two elements. We plotted them on separate plots, as is most commonly done on trading platforms.

There's more...

`TA-Lib` is a great library and the gold standard when it comes to calculating technical indicators. However, there are also alternative libraries out there, which are gaining traction. One of them is called `ta`. Compared to `TA-Lib`, which is a wrapper around a C++ library, `ta` is written using `pandas`, which makes exploring the code base much easier.

While it does not offer as extensive functionalities as `TA-Lib`, one of its unique features is that it can calculate all of the available 30+ indicators in a single line of code. That can definitely be useful in situations in which we want to calculate a lot of potential features for a machine learning model.

Execute the following steps to calculate 30+ technical indicators with a single line of code:

1. Import the libraries:

```
from ta import add_all_ta_features
```

2. Discard the previously calculated indicators and keep only the required columns:

```
df = df[["Open", "High", "Low", "Close", "Volume"]].copy()
```

3. Calculate all the technical indicators available in the `ta` library:

```
df = add_all_ta_features(df, open="Open", high="High",
                         low="Low", close="Close",
                         volume="Volume")
```

The resulting DataFrame contains 88 columns, out of which 83 were added by the single function call.

See also

Please find below links to repositories of TA-Lib, ta, and other interesting libraries useful for technical analysis:

- <https://github.com/mrjbq7/ta-lib>—The GitHub repository of TA-lib. Please refer to this source for more details on installing the library.
- <https://ta-lib.org/>
- <https://github.com/bukosabino/ta>
- <https://github.com/twopirllc/pandas-ta>
- <https://github.com/peerchemist/finta>

Downloading the technical indicators

We have already mentioned in *Chapter 1, Acquiring Financial Data*, that some data vendors not only provide historical stock prices but also offer a selection of the most popular technical indicators. In this recipe, we will show how to download the RSI indicator for IBM's stock, which can be directly compared to the one we calculated in the previous recipe using the TA-Lib library.

How to do it...

Execute the following steps to download the RSI calculated for IBM from Alpha Vantage:

1. Import the libraries:

```
from alpha_vantage.techindicators import TechIndicators
```

2. Instantiate the TechIndicators class and authenticate:

```
ta_api = TechIndicators(key="YOUR_KEY_HERE",
                        output_format="pandas")
```

3. Download the RSI for IBM's stock:

```
rsi_df, rsi_meta = ta_api.get_rsi(symbol="IBM",
                                    time_period=14)
```

4. Plot the downloaded RSI:

```
fig, ax = plt.subplots()
rsi_df.plot(ax=ax,
            title="RSI downloaded from Alpha Vantage")
ax.hlines(y=30,
           xmin=rsi_df.index.min(),
           xmax=rsi_df.index.max(),
           color="red")
```

```
ax.hlines(y=70,  
          xmin=rsi_df.index.min(),  
          xmax=rsi_df.index.max(),  
          color="red")
```

Running the snippet generates the following plot:

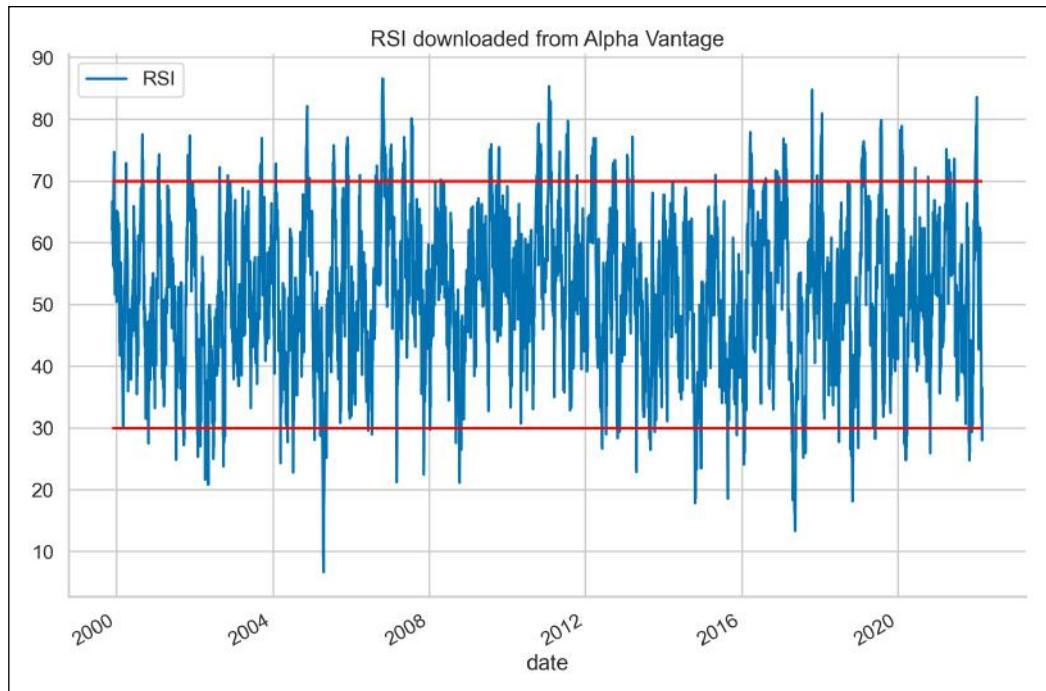


Figure 5.5: The RSI downloaded for IBM's stock prices

The downloaded DataFrame contains RSI values from November 1999 until the most recent date.

5. Explore the metadata object:

```
rsi_meta
```

By displaying the metadata object, we can see the following details of our request:

```
{'1: Symbol': 'IBM',  
'2: Indicator': 'Relative Strength Index (RSI)',  
'3: Last Refreshed': '2022-02-25',  
'4: Interval': 'daily',  
'5: Time Period': 14,  
'6: Series Type': 'close',  
'7: Time Zone': 'US/Eastern Time'}
```

How it works...

After importing the libraries, we instantiated the `TechIndicators` class, which can be used for downloading any of the available technical indicators (via the class's methods). While doing so, we provided our API key and indicated that we would like to receive the output in the form of a `pandas DataFrame`.

In *Step 3*, we downloaded the RSI for IBM's stock using the `get_rsi` method. At this point, we specified we wanted to use the last 14 days for calculating the indicator.



One thing to keep in mind while downloading calculated indicators is the data vendors' pricing policy. At the time of writing, the RSI endpoint of Alpha Vantage is a free one, while the MACD is a premium endpoint and requires purchasing a paid plan.

What can be a bit surprising is that we cannot specify the range of dates that we are interested in. We can clearly see this in *Step 4*, in which we can see data points going as far back as November 1999. We also plotted the RSI line, just as we have done in the previous recipe.

In the last step, we explored the metadata of the request, which contains the RSI's parameters, the stock symbol we requested, the latest refresh date, and which price series was used for calculating the indicator (in this case, the close price).

There's more...

Alpha Vantage is not the only data vendor that is providing access to technical indicators. Another one is Instruio. We demonstrate below how to download the MACD using its API:

1. Import the libraries:

```
import intrinio_sdk as intrinio
import pandas as pd
```

2. Authenticate using the personal API key and select the API:

```
intrinio.ApiClient().set_api_key("YOUR_KEY_HERE")
security_api = intrinio.SecurityApi()
```

3. Request the MACD for IBM's stock from 2020:

```
r = security_api.get_security_price_technicals_macd(  
    identifier="IBM",  
    fast_period=12,  
    slow_period=26,  
    signal_period=9,  
    price_key="close",  
    start_date="2020-01-01",  
    end_date="2020-12-31",  
    page_size=500  
)
```

While using Intrinio, we can actually specify the period for which we would like to download the indicator.

4. Convert the request's output into a pandas DataFrame:

```
macd_df = (  
    pd.DataFrame(r.technical_dict)  
    .sort_values("date_time")  
    .set_index("date_time")  
)  
macd_df.index = pd.to_datetime(macd_df.index).date
```

5. Plot the MACD:

```
fig, ax = plt.subplots(2, 1, sharex=True)  
  
(  
    macd_df[["macd_line", "signal_line"]]  
    .plot(ax=ax[0],  
          title="MACD downloaded from Intrinio")  
)  
  
ax[1].bar(df.index, macd_df["macd_histogram"].values,  
           label="macd_hist")  
ax[1].legend()
```

Running the snippet generates the following plot:

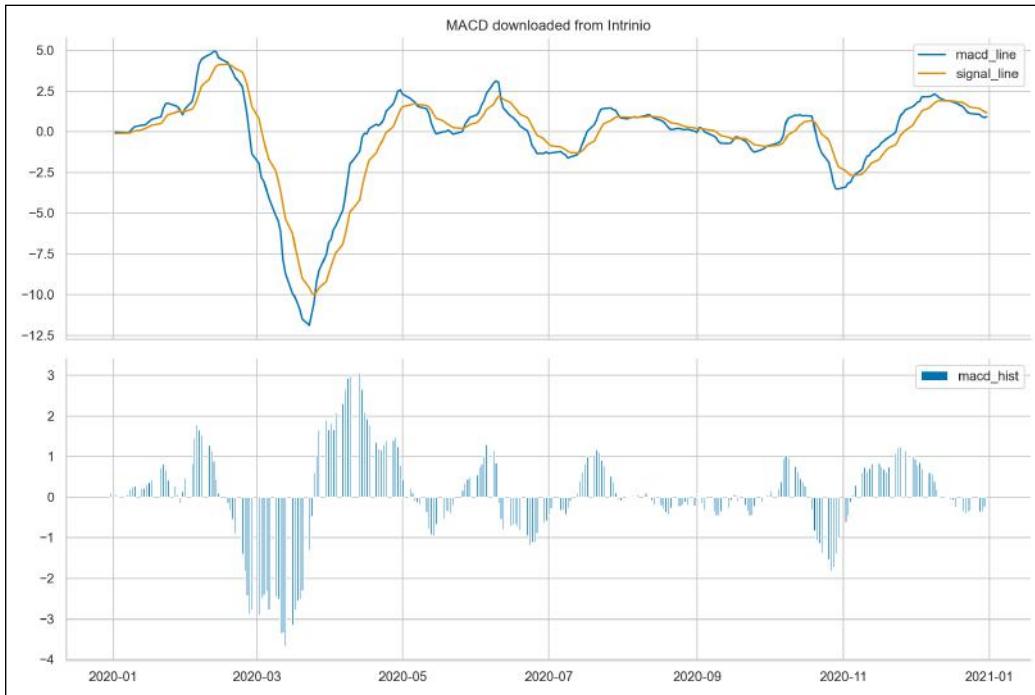


Figure 5.6: The MACD downloaded for IBM's stock prices

Recognizing candlestick patterns

In this chapter, we have already covered some of the most popular technical indicators. Another field of technical analysis that can be used for making trading decisions is **candlestick pattern recognition**. Overall, there are hundreds of candlestick patterns that can be used for determining the price direction and momentum.



Similar to all approaches to technical analysis, we should have a few things in mind while using pattern recognition. First, the patterns are only informative within the limitations of the given chart (in a specified frequency: intraday, daily, weekly, and so on). Second, the patterns' predictive potency decreases very quickly after a few (3–5) bars once the pattern has been completed. Third, in the modern electronic environment, many signals identified by analyzing candlestick patterns might not work reliably anymore. Some big players are also able to set up traps by creating fake candlestick patterns to be picked up by other market participants.

Bulkowski (2021) divides the patterns into two categories, based on the expected outcomes:

- Reversal patterns—such patterns predict a change in the price's direction
- Continuation patterns—such patterns predict an extension in the current trend

In this recipe, we try to identify the **three line strike** pattern in hourly Bitcoin prices. That pattern belongs to the continuation group. Its bearish variant (identified in an overall bearish trend) is characterized by three bars, each one having a lower low than the previous one. The fourth bar of the pattern opens at the third candle's low or even lower, but then reverses heavily and closes above the high of the first candle of the series.

How to do it...

Execute the following steps to identify the three line strike pattern in Bitcoin's hourly candlesticks:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
import talib
import mplfinance as mpf
```

2. Download Bitcoin's hourly prices from the last 9 months:

```
df = yf.download("BTC-USD",
                  period="9mo",
                  interval="1h",
                  progress=False)
```

3. Identify the three line strike pattern:

```
df["3_line_strike"] = talib.CDL3LINESTRIKE(
    df["Open"], df["High"], df["Low"], df["Close"]
)
```

4. Locate and plot the bearish pattern:

```
df[df["3_line_strike"] == -100].head()
```

	Open	High	Low	Close	Adj Close	Volume	3_line_strike
2021-06-06 14:00:00+00:00	35655.64	36185.68	35544.90	36124.10	36124.10	131508224	-100
2021-06-13 15:00:00+00:00	35807.91	36082.75	35807.91	36082.75	36082.75	0	-100
2021-07-03 17:00:00+00:00	34546.72	34909.26	34546.72	34793.32	34793.32	24389632	-100
2021-07-08 19:00:00+00:00	32843.35	33088.62	32657.65	33013.81	33013.81	343470080	-100
2021-07-16 12:00:00+00:00	31115.04	31703.58	31115.04	31703.58	31703.58	351070208	-100

Figure 5.7: The first five observations of the bearish three line strike pattern

```
mpf.plot(df["2021-07-16 05:00:00":"2021-07-16 16:00:00"],
          type="candle")
```

Executing the snippet returns the following plot:

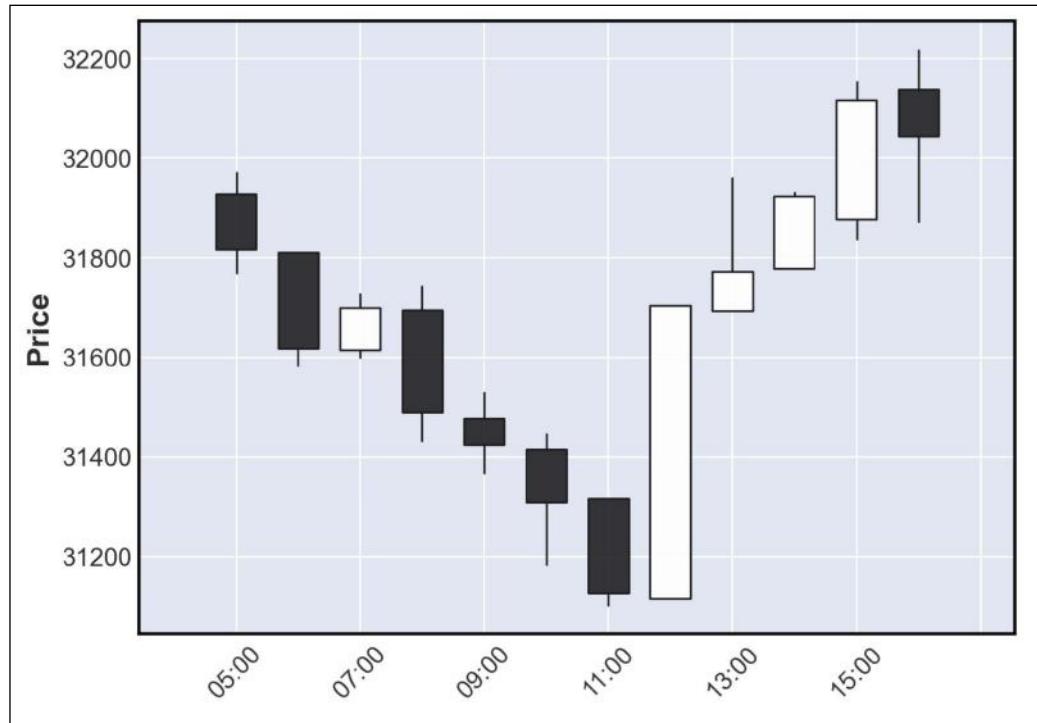


Figure 5.8: The identified bearish three line strike pattern

- Locate and plot the bullish pattern:

```
df[df["3_line_strike"] == 100]
```

	Open	High	Low	Close	Adj Close	Volume	3_line_strike
2021-06-08 06:00:00+00:00	32932.72	32971.01	32620.67	32637.23	32637.23	359575552	100
2021-06-30 08:00:00+00:00	35244.68	35279.57	34594.18	34654.98	34654.98	0	100
2021-07-02 01:00:00+00:00	33775.50	33775.50	33276.16	33276.16	33276.16	0	100
2021-07-05 11:00:00+00:00	34478.20	34491.51	33429.94	33461.20	33461.20	1175277568	100
2021-07-10 17:00:00+00:00	34041.45	34041.45	33324.12	33432.79	33432.79	0	100

Figure 5.9: The first five observations of the bullish three line strike pattern

```
mpf.plot(df["2021-07-10 10:00:00":"2021-07-10 23:00:00"],
          type="candle")
```

Executing the snippet returns the following plot:

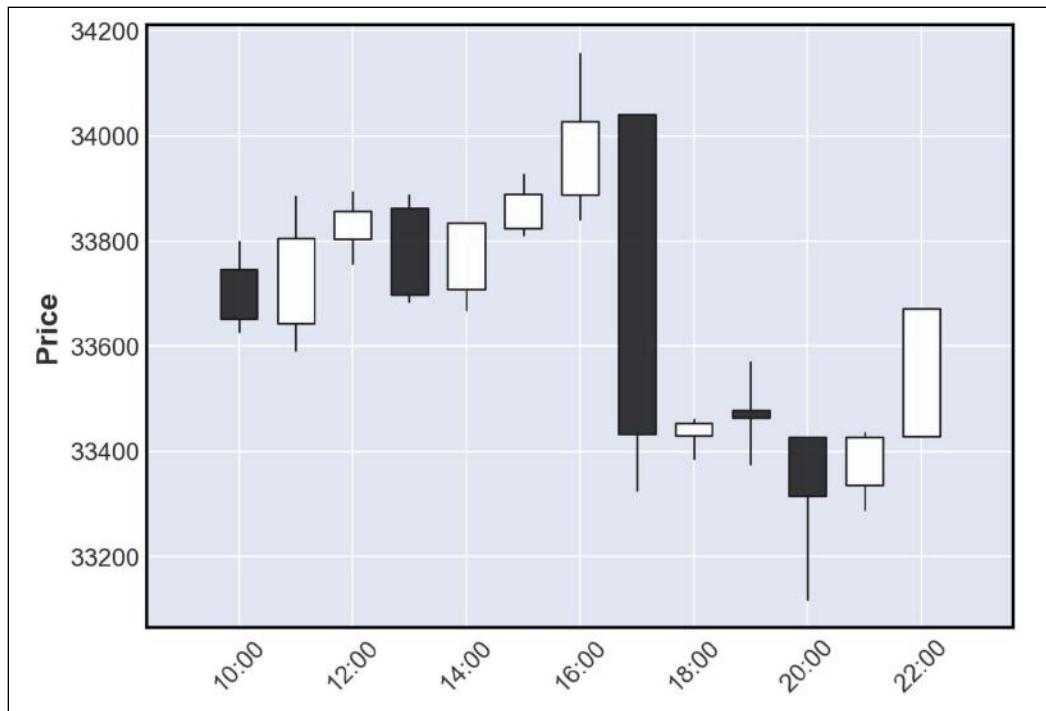


Figure 5.10: The identified bullish three line strike pattern

We could use the identified patterns to create trading strategies. For example, a bearish three line strike usually signals a small pullback that will be followed by a continuation of the bearish trend.

How it works...

After importing the libraries, we downloaded hourly Bitcoin prices from the last 3 months using the `yfinance` library.

In *Step 3*, we used the `TA-Lib` library to identify the three line strike pattern (with the `CDL3LINESTRIKE` function). We had to separately provide the OHLC prices as inputs for the function. We store the outputs of the function in a new column. For this function, there are three possible outputs:

- 100—Indicates the bullish variant of the pattern
- 0—No pattern detected
- -100—Indicates the bearish variant of the pattern

The authors of the library warn that the user should consider the three line strike pattern to be significant when it appears in a trend in the same direction (this is not verified by the library).



Certain functions can have additional possible outputs. Some of them also have values of -200/200 (for example, for the Hikkake pattern), whenever there is some additional confirmation in the pattern.

In *Step 4*, we filtered the DataFrame for a bearish pattern. It has been identified six times and we chose the one from 2021-07-16 12:00:00. Then, we plotted the pattern together with some neighboring candles.

In *Step 5*, we have repeated the same procedure, this time for a bullish pattern.

There's more...

If we wanted to use the identified patterns as features for a model/strategy, it might be worthwhile to try to identify all the possible patterns at once. We can do so by executing the following steps:

1. Get all available pattern names:

```
candle_names = talib.get_function_groups()["Pattern Recognition"]
```

2. Iterate over the list of patterns and try identifying them all:

```
for candle in candle_names:
    df[candle] = getattr(talib, candle)(df["Open"], df["High"],
                                         df["Low"], df["Close"])
```

3. Inspect the summary statistics of the patterns:

```
with pd.option_context("display.max_rows", len(candle_names)):
    display(df[candle_names].describe().transpose().round(2))
```

For brevity, we only present the top 10 rows of the returned DataFrame:

	count	mean	std	min	25%	50%	75%	max
CDL2CROWS	6454.0	-0.11	3.29	-100.0	0.0	0.0	0.0	0.0
CDL3BLACKCROWS	6454.0	-0.06	2.49	-100.0	0.0	0.0	0.0	0.0
CDL3INSIDE	6454.0	0.03	10.99	-100.0	0.0	0.0	0.0	100.0
CDL3LINESTRIKE	6454.0	0.08	6.70	-100.0	0.0	0.0	0.0	100.0
CDL3OUTSIDE	6454.0	0.26	23.39	-100.0	0.0	0.0	0.0	100.0
CDL3STARSISSOUTH	6454.0	0.00	0.00	0.0	0.0	0.0	0.0	0.0
CDL3WHITE SOLDIERS	6454.0	0.19	4.31	0.0	0.0	0.0	0.0	100.0
CDLABANDONEDBABY	6454.0	0.00	0.00	0.0	0.0	0.0	0.0	0.0
CDLADVANCEBLOCK	6454.0	-1.30	11.33	-100.0	0.0	0.0	0.0	0.0
CDLBELTHOLD	6454.0	-0.20	42.63	-100.0	0.0	0.0	0.0	100.0

Figure 5.11: Summary statistics of the identified candlestick patterns

We can see that some patterns were never identified (the minimum and maximum of zero), while others had either one or two of the variants (bullish or bearish). In the notebook (available on GitHub), we have also looked into identifying the **evening star** pattern based on the outputs of this table.

See also

- https://sourceforge.net/p/ta-lib/code/HEAD/tree/trunk/ta-lib/c/src/ta_func/
- Bulkowski, T. N. 2021 *Encyclopedia of Chart Patterns*. John Wiley & Sons, 2021.

Building an interactive web app for technical analysis using Streamlit

In this chapter, we have already covered the basics of technical analysis, which can help traders make their decision. However, until now everything was quite static—we downloaded the data, calculated an indicator, plotted it, and if we wanted to change the asset or the range of dates, we had to repeat all the steps. What if there was a better and more interactive way to approach this challenge?

This is exactly where Streamlit comes into play. Streamlit is an open source framework (and a company under the same name, similarly to Plotly) that allows us to build interactive web apps using only Python, all within minutes. Below you can find the highlights of Streamlit:

- It is easy to learn and can generate results very quickly
- It is Python only; no frontend experience is required
- It allows us to focus purely on the data/ML sides of the app
- We can use Streamlit's hosting services for our apps

In this recipe, we will build an interactive app used for technical analysis. You will be able to select any of the constituents of the S&P 500 and carry out a simple analysis quickly and in an interactive way. What is more, you can easily expand the app to add more features such as different indicators and assets, or even embed backtesting of trading strategies within the app.

Getting ready

This recipe is slightly different than the rest. The code of our app “lives” in a single Python script (`technical_analysis_app.py`), which has around a hundred lines of code. A very basic app can be much more concise, but we wanted to go over some of the most interesting features of Streamlit, even if they are not strictly necessary to make a basic app for technical analysis.

In general, Streamlit executes code from top to bottom, which makes the explanation easier to fit into the structure used in this book. Thus, the steps in this recipe are not steps *per se*—they cannot/should not be executed on their own. Instead, they are a step-by-step walkthrough of all the components of the app. While building your own apps or expanding this one, you can freely change the order of the steps as you see fit (as long as they are aligned with Streamlit's framework).

How to do it...

The following steps are all located in the `technical_analysis_app.py`:

1. Import the libraries:

```
import yfinance as yf
import streamlit as st
import datetime
import pandas as pd
import cufflinks as cf
from plotly.offline import iplot

cf.go_offline()
```

2. Define a function for downloading a list of S&P 500 constituents from Wikipedia:

```
@st.cache
def get_sp500_components():
    df = pd.read_html("https://en.wikipedia.org/wiki/List_of_S%26P_500_
    companies")
    df = df[0]
    tickers = df["Symbol"].to_list()
    tickers_companies_dict = dict(
        zip(df["Symbol"], df["Security"]))
    )
    return tickers, tickers_companies_dict
```

3. Define a function for downloading historical stock prices using `yfinance`:

```
@st.cache
def load_data(symbol, start, end):
    return yf.download(symbol, start, end)
```

4. Define a function for storing downloaded data as a CSV file:

```
@st.cache
def convert_df_to_csv(df):
    return df.to_csv().encode("utf-8")
```

5. Define the part of the sidebar used for selecting the ticker and the dates:

```
st.sidebar.header("Stock Parameters")

available_tickers, tickers_companies_dict = get_sp500_components()
```

```
ticker = st.sidebar.selectbox(  
    "Ticker",  
    available_tickers,  
    format_func=tickers_companies_dict.get  
)  
start_date = st.sidebar.date_input(  
    "Start date",  
    datetime.date(2019, 1, 1)  
)  
end_date = st.sidebar.date_input(  
    "End date",  
    datetime.date.today()  
)  
  
if start_date > end_date:  
    st.sidebar.error("The end date must fall after the start date")
```

6. Define the part of the sidebar used for tuning the details of the technical analysis:

```
st.sidebar.header("Technical Analysis Parameters")  
  
volume_flag = st.sidebar.checkbox(label="Add volume")
```

7. Add the expander with parameters of the SMA:

```
exp_sma = st.sidebar.expander("SMA")  
sma_flag = exp_sma.checkbox(label="Add SMA")  
sma_periods= exp_sma.number_input(  
    label="SMA Periods",  
    min_value=1,  
    max_value=50,  
    value=20,  
    step=1  
)
```

8. Add the expander with parameters of the Bollinger bands:

```
exp_bb = st.sidebar.expander("Bollinger Bands")  
bb_flag = exp_bb.checkbox(label="Add Bollinger Bands")  
bb_periods= exp_bb.number_input(label="BB Periods",  
    min_value=1, max_value=50,  
    value=20, step=1)  
bb_std= exp_bb.number_input(label="# of standard deviations",  
    min_value=1, max_value=4,  
    value=2, step=1)
```

9. Add the expander with parameters of the RSI:

```
exp_rsi = st.sidebar.expander("Relative Strength Index")
rsi_flag = exp_rsi.checkbox(label="Add RSI")
rsi_periods= exp_rsi.number_input(
    label="RSI Periods",
    min_value=1,
    max_value=50,
    value=20,
    step=1
)
rsi_upper= exp_rsi.number_input(label="RSI Upper",
                                 min_value=50,
                                 max_value=90, value=70,
                                 step=1)
rsi_lower= exp_rsi.number_input(label="RSI Lower",
                                 min_value=10,
                                 max_value=50, value=30,
                                 step=1)
```

10. Specify the title and additional text in the app's main body:

```
st.title("A simple web app for technical analysis")
st.write("""
    ### User manual
    * you can select any company from the S&P 500 constituents
""")
```

11. Load the historical stock prices:

```
df = load_data(ticker, start_date, end_date)
```

12. Add the expander with a preview of the downloaded data:

```
data_exp = st.expander("Preview data")
available_cols = df.columns.tolist()
columns_to_show = data_exp.multiselect(
    "Columns",
    available_cols,
    default=available_cols
)
```

```
data_exp.dataframe(df[columns_to_show])

csv_file = convert_df_to_csv(df[columns_to_show])
data_exp.download_button(
    label="Download selected as CSV",
    data=csv_file,
    file_name=f"{ticker}_stock_prices.csv",
    mime="text/csv",
)
```

13. Create the candlestick chart with the selected TA indicators:

```
title_str = f"{tickers_companies_dict[ticker]}'s stock price"
qf = cf.QuantFig(df, title=title_str)
if volume_flag:
    qf.add_volume()
if sma_flag:
    qf.add_sma(periods=sma_periods)
if bb_flag:
    qf.add_bollinger_bands(periods=bb_periods,
                           boll_std=bb_std)
if rsi_flag:
    qf.add_rsi(periods=rsi_periods,
               rsi_upper=rsi_upper,
               rsi_lower=rsi_lower,
               showbands=True)

fig = qf.iplot(asFigure=True)
st.plotly_chart(fig)
```

To run the app, open the terminal, navigate to the directory in which the `technical_analysis_app.py` script is located, and run the following command:

```
streamlit run technical_analysis_app.py
```

Running the code opens the Streamlit app in your default browser. The app's default screen looks as follows:

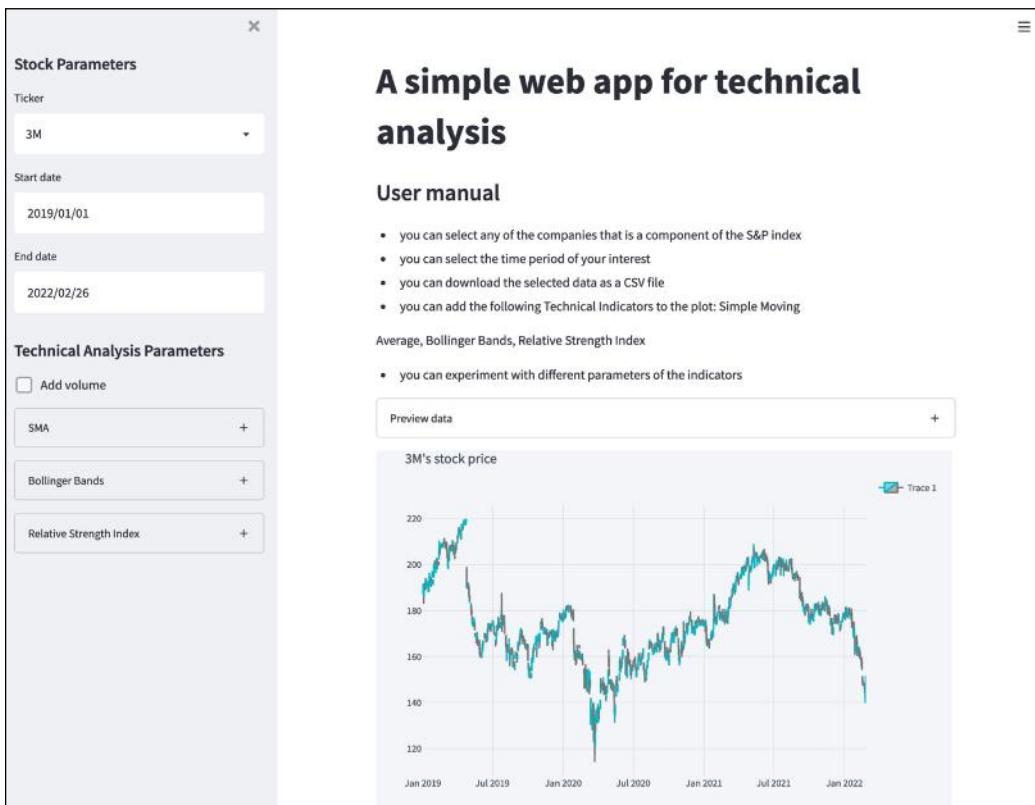


Figure 5.12: Our technical analysis app in the browser

The app is fully responsive to the inputs—anytime you change the inputs in the sidebar or the app's main body, the displayed contents will adjust accordingly. Potentially, we could even take it a step further and connect our app to a broker via the broker's API. This way, we could analyze the patterns in the app and create orders based on the outcome of our analyses.

How it works...

As mentioned in the *Getting ready* section, this recipe is structured differently. The steps are in fact a sequence of elements that all define the app we built. Before diving into the details, the general structure of the app's codebase is as follows:

- Imports and setup (*step 1*)
- Data loading functions (*steps 2–4*)
- Sidebar (*steps 5–9*)
- App's main body (*steps 10–13*)

In the first step, we imported the required libraries. For the technical analysis part, we decided to use a library that can visualize a selection of technical indicators in as few lines of code as possible. That is why we decided to go with `cufflinks`, which was introduced in *Chapter 3, Visualizing Financial Time Series*. However, in case you need to calculate a wider range of indicators, you can use any other library and create the plots yourself.

In *Step 2*, we defined a function for loading a list of S&P 500 constituents from Wikipedia. We used the `pd.read_html` to download the information from the table straight into a DataFrame. The function returns two elements: a list of valid tickers and a dictionary containing pairs of tickers and their corresponding companies' names.

You surely have noticed that we used a `@st.cache` decorator while defining the function. We will not go over a lot of details of decorators in general, but we will cover what this one does as it is very handy while building an app using Streamlit. The decorator indicates that the app should cache the previously fetched data for later use. So in case we refresh the page or the function is called again, the data will not be downloaded/processed again (unless some conditions occur). This way, we can greatly increase the web app's responsiveness and lower the end user's waiting time.

Behind the scenes, Streamlit keeps track of the following information to determine if the data should be fetched again:

- The input parameters that we provided while calling the function
- The values of any external variables used in the function
- The body of the called function
- The body of any function called inside of the cached function

In short, if this is the first time Streamlit sees a certain combination of those four elements, it will execute the function and store its output in a local cache. If it encounters the very same set of items the next time the function is called, it will skip executing it and return the cached output from the previous execution.

Steps 3 and 4 contain very small functions. The first one is used to fetch the historical stock prices from Yahoo Finance using the `yfinance` library. The following step saves the output of a DataFrame into a CSV file, which is then encoded in UTF-8.

In *Step 5*, we started working on the app's sidebar, which we use for storing the parameter configurations for the app. The first thing to notice is that all the elements that are meant to be located in the sidebar are called with `st.sidebar` (as opposed to just `st`, which we use when defining the main body's elements and other functions). In this step, we did the following:

- We specified the header.
- We downloaded the list of available tickers.
- We created a drop-down selection box of the available tickers. We also provided additional formatting by passing the dictionary containing symbol-name pairs to the `format_func` argument.
- We allowed the users to select the start and end dates for the analysis. Using `date_input` displays an interactive calendar from which the users can select a date.

- We accounted for invalid combinations of dates (start later than the end) by using an `if` statement together with `st.sidebar.error`. This will halt the app execution until the error is resolved, that is, until a proper input is provided.

The outcome of this step looks as follows:

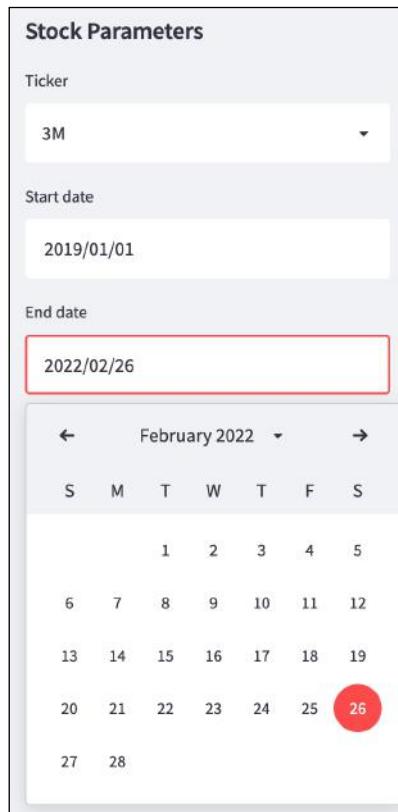
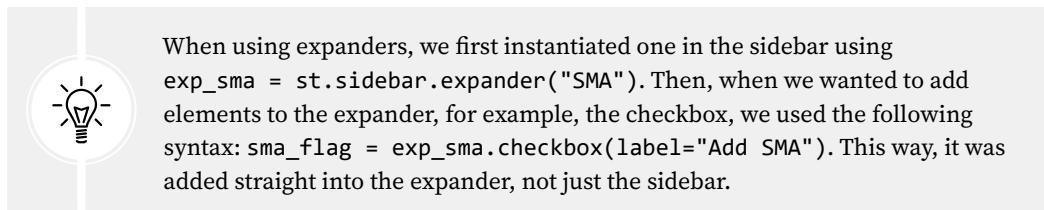


Figure 5.13: Part of the sidebar where we can choose the ticket and start/end dates

In *Step 6*, we added another header to our sidebar and created a checkbox using `st.checkbox`. If checked, the assigned variable will hold a `True` value, `False` if unchecked.

In *Step 7*, we started with configuring the technical indicators. To keep the app clean, we used expanders (`st.expander`). Expanders are collapsible boxes, which we trigger to expand by pressing the plus icon. Inside, we stored two elements:

- A checkbox indicating whether we want to display the SMA.
- A numeric field in which we can specify the number of periods for the moving average. For that element, we used Streamlit's `number_input` object. We provided the label, minimum/maximum values, the default value, and the step size (we can incrementally increase/decrease the value of the field by that number when we press the corresponding buttons).



When using expanders, we first instantiated one in the sidebar using `exp_sma = st.sidebar expander("SMA")`. Then, when we wanted to add elements to the expander, for example, the checkbox, we used the following syntax: `sma_flag = exp_sma.checkbox(label="Add SMA")`. This way, it was added straight into the expander, not just the sidebar.

Steps 8 and 9 are very similar. We created two expanders for other technical indicators we wanted to include in the app – Bollinger bands and the RSI.

The code from Steps 7–9 generates the following part of the app's sidebar:

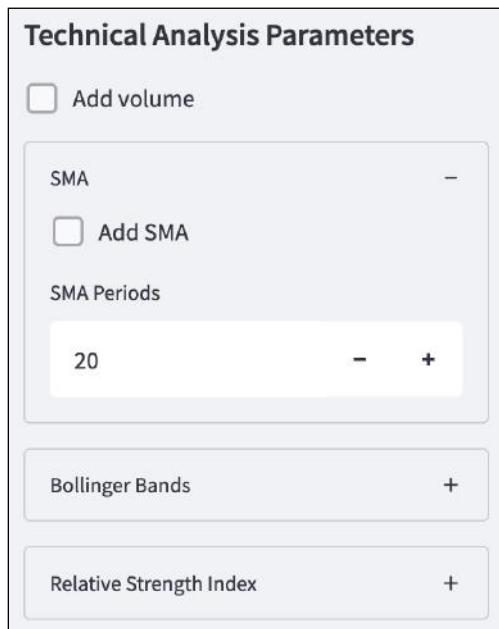


Figure 5.14: Part of the sidebar where we can modify the parameters of the selected indicators

Then, we proceeded to define the app's main body. In Step 10, we added the app's title using `st.title` and added a user manual using `st.write`. When using the latter function, we can provide text input in a Markdown-formatted string. For this part, we used a subheader (indicated by `###`) and created a list of bullets indicated by `*`. For brevity, we did not include all the text in the book, but you can find it in the book's GitHub repository.

In Step 11, we downloaded the historical stock prices based on the inputs from the sidebar. What we could also have done here is download a full range of dates available for a given stock and only then use the sidebar's start/end dates to filter out the periods of interest. By doing so, we would not have to redownload the data anytime we changed the start/end dates.

In Step 12, we defined another expander, this time in the app's main body. First, we added a multiple selection field (`st.multiselect`) from which we can select any of the available columns from the downloaded historical prices. Then, we displayed the selected columns of the DataFrame for further inspection using `st.dataframe`. Lastly, we added the functionality to download the selected data (including the column selection) as a CSV file. For that, we used our `convert_df_to_csv` function, together with `st.download_button`.

Step 12 is responsible for generating the following part of the app:

The screenshot shows a Jupyter Notebook cell with the title "Preview data". Inside the cell, there is a pandas DataFrame titled "Columns" with the following data:

	Open	High	Low	Close	Adj Close	Volume
2018-12-31T00:00:00	190.3400	191.6500	188.5000	190.5400	170.5956	
2019-01-02T00:00:00	187.8200	190.9900	186.7000	190.9500	170.9627	
2019-01-03T00:00:00	188.2800	188.2800	182.8900	183.7600	164.5253	
2019-01-04T00:00:00	186.7500	191.9800	186.0300	191.3200	171.2940	
2019-01-07T00:00:00	191.3600	192.3000	188.6600	190.8800	170.9000	
2019-01-08T00:00:00	193.0000	194.1100	189.5800	191.6800	171.6163	
2019-01-09T00:00:00	193.2500	193.9400	191.3800	192.3000	172.1714	
2019-01-10T00:00:00	190.8700	193.8100	189.4000	193.6000	173.3353	
2019-01-11T00:00:00	191.8400	192.6900	190.8600	192.2100	172.0909	
2019-01-14T00:00:00						

At the bottom of the cell, there is a button labeled "Download selected as CSV".

Figure 5.15: Part of the app where we can inspect the DataFrame containing prices and download it as a CSV

In the app's last step, we defined the figure we wanted to display. Without any of the technical analysis inputs, the app will display a candlestick chart using `cufflinks`. We instantiated the `QuantFig` object and then added elements to it depending on the inputs from the sidebar. Each of the Boolean flags triggers a separate command that adds an element to the plot. To display the interactive figure, we used `st.plotly_chart`, which works with `plotly` figures (`cufflinks` is a wrapper on top of `plotly`).



For other visualization libraries, there are different commands to embed visualizations. For example, for `matplotlib`, we would use `st.pyplot`. We could also display plots created in Altair using `st.altair_chart`.

There's more...

In the first edition of the book, we covered a bit of a different approach to creating an interactive dashboard for technical analysis. Instead of Streamlit, we used `ipywidgets` to build the dashboard inside of a Jupyter notebook.

In general, Streamlit might be the better tool for this particular job, especially if we want to deploy the app (covered in the next recipe) and share it with others. However, `ipywidgets` can still be useful for other projects, which can live locally inside of a notebook. That is why you can find the code used for creating a very similar dashboard (within a notebook) in the accompanying GitHub repository.

See also

- <https://streamlit.io/>
- <https://docs.streamlit.io/>

Deploying the technical analysis app

In the previous recipe, we created a full-fledged web app for technical analysis, which we can easily run and use locally. However, that is not always the goal, as we might want to access the app from anywhere or share it with our friends or colleagues. That is why the next step would be to deploy the app to the cloud.

In this recipe, we show how to deploy the app using Streamlit's (the company) services.

Getting ready

To deploy the app to Streamlit Cloud, we need to create an account there (<https://forms.streamlit.io/community-sign-up>). You will also need a GitHub account to host the code of the app.

How to do it...

Execute the following steps to deploy the Streamlit app to the cloud:

1. Host the code base of the app in a public repository on GitHub:

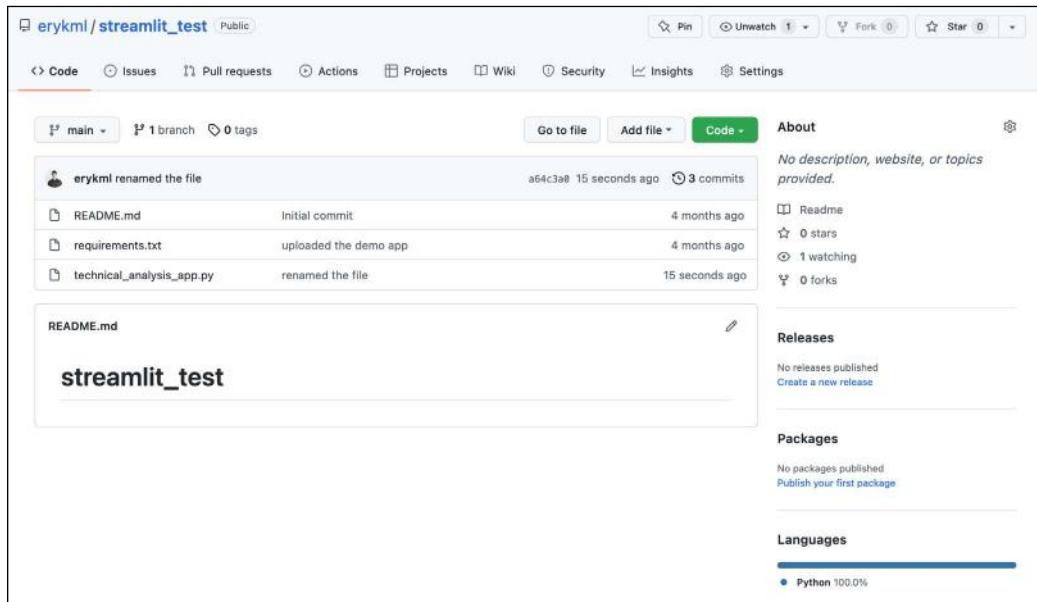


Figure 5.16: The code base of the app hosted in a public GitHub repository

In this step, remember to host the entire code base of the app, which can naturally be spread out over multiple files. Also, please include some kind of a dependencies list. In our case, it is the requirements.txt file.

2. Go to <https://share.streamlit.io/> and log in. You might need to connect your GitHub account to your Streamlit account and authorize it to have certain types of access to your GitHub account.
3. Click the *New app* button.
4. Provide the required details: the name of the repository in your profile, the branch, and the file containing the app:

Deploy an app

Apps are deployed directly from their GitHub repo. Enter the location of your app below.

Repository	Paste GitHub URL
erykml/streamlit_test	
Branch	main
Main file path	technical_analysis_app.py
Advanced settings...	
Deploy!	

Figure 5.17: The information we have to provide to create the app

5. Click *Deploy!*.

Now, you can go to the provided link to use the app.

How it works...

In the first step, we hosted the app's code in a public GitHub repository. If you are new to Git or GitHub, please refer to the link in the *See also* section for more information. At the time of writing, it is not possible to use other version control providers such as GitLab or BitBucket for hosting code of Streamlit apps. The bare minimum in terms of the files is the app's script (in our case, `technical_analysis_app.py`) and some form of a list with requirements. The easiest one would be a simple `requirements.txt` text file containing all the libraries you would like to use in the app. If you are using a different dependency manager (`conda`, `pipenv`, or `poetry`), you need to provide their respective files.



If there are multiple libraries you would like to use in the app, the easiest way to create the requirements file containing them would be to run `pip freeze > requirements.txt` while having your virtual environment activated.

All the next steps are quite intuitive, as Streamlit's platform is very easy to navigate. What might be useful to mention is that in *Step 4*, we can also provide some more advanced settings. They include:

- The Python version that you would like the app to use.
- The Secrets field, in which you can store some environment variables and secrets, such as API keys. In general, it is against best practices to store usernames, API keys, and other secrets in public GitHub repositories. If your app is fetching data from some provider or your internal database, that is the field in which you can safely store the credentials, which will be then encrypted and served securely to your app at runtime.

There's more...

In this recipe, we showed how to deploy our web app to the Streamlit Cloud. While being the simplest, it is not the only option. Another one would be to deploy the app to Heroku, which is a **Platform as a Service (PaaS)** type of platform that enables you to build, run, and operate applications entirely in the cloud.

See also

- <https://www.heroku.com/>—for more information about Heroku's services
- <https://docs.streamlit.io/streamlit-cloud>—for more details on how to deploy the app and what are the best practices
- <https://docs.github.com/en/get-started/quickstart/hello-world>—a tutorial on how to use GitHub

Summary

In this chapter, we have learned about technical analysis. We started by calculating some of the most popular technical indicators (and downloading precalculated ones): the SMA, the RSI, and the MACD. We have also explored identifying patterns in candlesticks. Lastly, we learned how to create and deploy an interactive app for technical analysis.

In further chapters, we will put this knowledge into practice by creating and backtesting trading strategies based on the technical indicators we have already covered.

6

Time Series Analysis and Forecasting

Time series are omnipresent in both industry and research. We can find examples of time series in commerce, tech, healthcare, energy, finance, and so on. We are mostly interested in the last one, as the time dimension is inherent to trading and many financial/economic indicators. However, pretty much every business generates some sort of time series, for example, its profits collected over time or any other measured KPI. That is why the techniques we cover in the following two chapters can be used for any time series analysis task you might encounter in your line of work.

Time series modeling or forecasting can often be approached from different angles. The two most popular are statistical methods and machine learning approaches. Additionally, we will also cover some examples of using deep learning for time series forecasting in *Chapter 15, Deep Learning in Finance*.

In the past, when we did not have vast computing power available at our disposal and the time series were not that granular (as data was not collected everywhere and all the time), statistical approaches dominated the domain. Recently, the situation has changed, and ML-based approaches are taking the lead when it comes to time series models running in production. However, that does not mean that the classical statistical approaches are not relevant anymore—in fact, far from it. They can still produce state-of-the-art results in cases when we have very little training data (for example, 3 years of monthly data) and the ML models simply cannot learn the patterns from it. Also, we can observe that statistical approaches were used to win quite a few of the most recent M-Competitions (the biggest time series forecasting competition started by Spyros Makridakis).

In this chapter, we introduce the basics of time series modeling. We start by explaining the building blocks of time series and how to separate them using decomposition methods. Later, we cover the concept of stationarity—why it is important, how to test for it, and ultimately, how to achieve it if the original series is not stationary.

Then, we look at two of the most widely used statistical approaches to time series modeling—exponential smoothing methods and ARIMA class models. In both cases, we show you how to fit the models, evaluate their goodness of fit, and forecast the future values of the time series.

We cover the following recipes in this chapter:

- Time series decomposition
- Testing for stationarity in time series
- Correcting for stationarity in time series
- Modeling time series with exponential smoothing methods
- Modeling time series with ARIMA class models
- Finding the best-fitting ARIMA model with auto-ARIMA

Time series decomposition

One of the goals of time series decomposition is to increase our understanding of the data by breaking down the series into multiple components. It provides insight in terms of modeling complexity and which approaches to follow in order to accurately capture/model each of the components.

An example can shed more light on the possibilities. We can imagine a time series with a clear trend, either increasing or decreasing. On one hand, we could use the decomposition to extract the trend component and remove it from our time series before modeling the remaining series. This could help with making the time series stationary (please refer to the following recipe for more details). Then, we can always add it back after the rest of the components have been accounted for. On the other hand, we could provide enough data or adequate features for our algorithm to model the trend itself.

The components of time series can be divided into two types: systematic and non-systematic. The systematic ones are characterized by consistency and the fact that they can be described and modeled. By contrast, the non-systematic ones cannot be modeled directly.

The following are the **systematic components**:

- **Level**—the mean value in the series.
- **Trend**—an estimate of the trend, that is, the change in value between successive time points at any given moment. It can be associated with the slope (increasing/decreasing) of the series. In other words, it is the general direction of the time series over a long period of time.
- **Seasonality**—deviations from the mean caused by repeating short-term cycles (with fixed and known periods).

The following is the non-systematic component:

- **Noise**—the random variation in the series. It consists of all the fluctuations that are observed after removing other components from the time series.

The classical approach to time series decomposition is usually carried out using one of two types of models: additive and multiplicative.

An **additive model** can be described by the following characteristics:

- Model's form— $y(t) = \text{level} + \text{trend} + \text{seasonality} + \text{noise}$
- Linear model—changes over time are consistent in size

- The trend is linear (straight line)
- Linear seasonality with the same frequency (width) and amplitude (height) of cycles over time

A **multiplicative model** can be described by the following characteristics:

- Model's form— $y(t) = \text{level} * \text{trend} * \text{seasonality} * \text{noise}$
- Non-linear model—changes over time are not consistent in size, for example, exponential
- A curved, non-linear trend
- Non-linear seasonality with increasing/decreasing frequency and amplitude of cycles over time

To make things more interesting, we can find time series with combinations of additive and multiplicative characteristics, for example, a series with additive trend and multiplicative seasonality.

Please refer to the following figure for visualization of the possible combinations. And while real-world problems are never that simple (noisy data with varying patterns), these abstract models offer a simple framework that we can use to analyze our time series before attempting to model/forecast it.

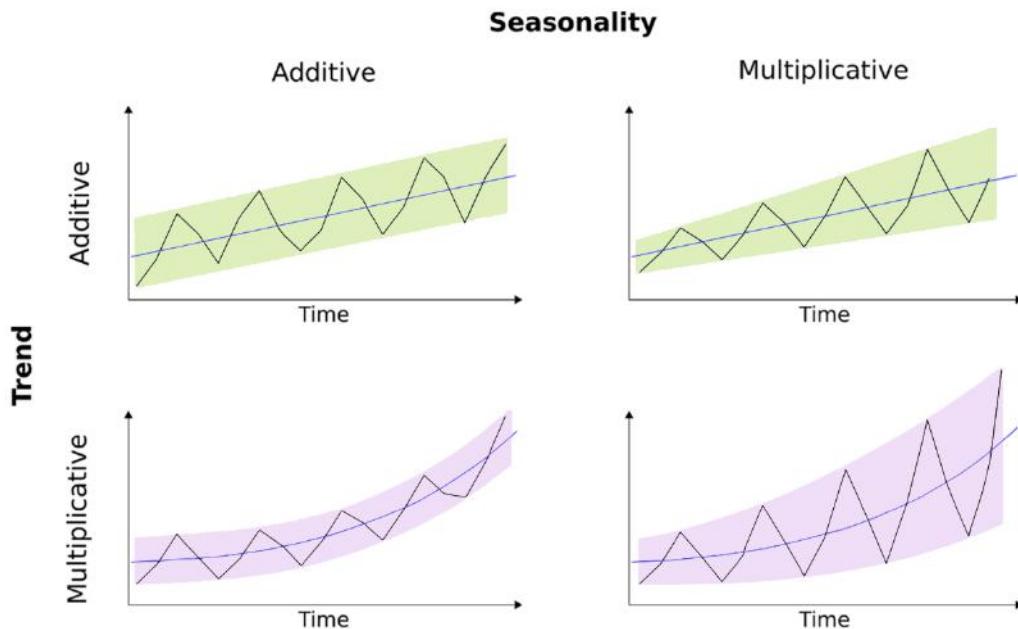


Figure 6.1: Additive and multiplicative variants of trend and seasonality

It can be the case that we do not want to (or simply cannot, due to some models' assumptions) work with a multiplicative model. One possible solution is to transform the multiplicative model into an additive one using logarithmic transformation:

$$\log(\text{time} * \text{seasonality} * \text{residual}) = \log(\text{time}) + \log(\text{seasonality}) + \log(\text{residual})$$

In this recipe, we will present how to carry out time series decomposition of monthly US unemployment rates downloaded from the Nasdaq Data Link.

How to do it...

Execute the following steps to carry out the time series decomposition:

1. Import the libraries and authenticate:

```
import pandas as pd
import nasdaqdatalink
import seaborn as sns
from statsmodels.tsa.seasonal import seasonal_decompose

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

2. Download the monthly US unemployment rate from the years 2010 to 2019:

```
df = (
    nasdaqdatalink.get(dataset="FRED/UNRATENSA",
                        start_date="2010-01-01",
                        end_date="2019-12-31")
    .rename(columns={"Value": "unemp_rate"})
)
```

In Figure 6.2, we can see some clear seasonal patterns in the time series. We did not include more recent data in this analysis, as the COVID-19 pandemic caused quite abrupt changes in any patterns observable in the unemployment rate time series. We do not show the code used for generating the plot, as it is very similar to the one used in *Chapter 3, Visualizing Financial Time Series*.

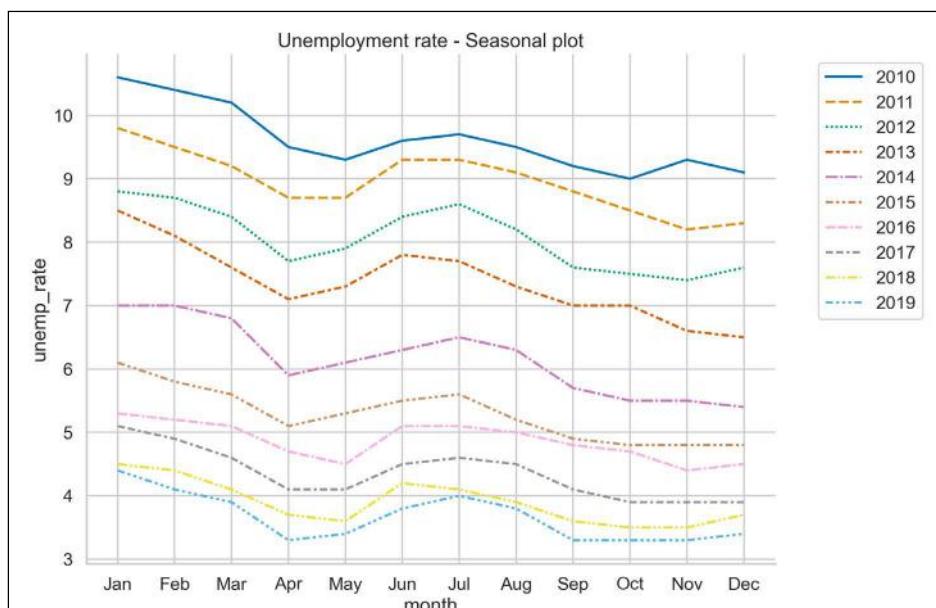


Figure 6.2: Seasonal plot of the US unemployment rate in the years 2010 to 2019

3. Add rolling mean and standard deviation:

```
WINDOW_SIZE = 12
df["rolling_mean"] = df["unemp_rate"].rolling(window=WINDOW_SIZE).mean()
df["rolling_std"] = df["unemp_rate"].rolling(window=WINDOW_SIZE).std()
df.plot(title="Unemployment rate")
```

Running the snippet generates the following plot:

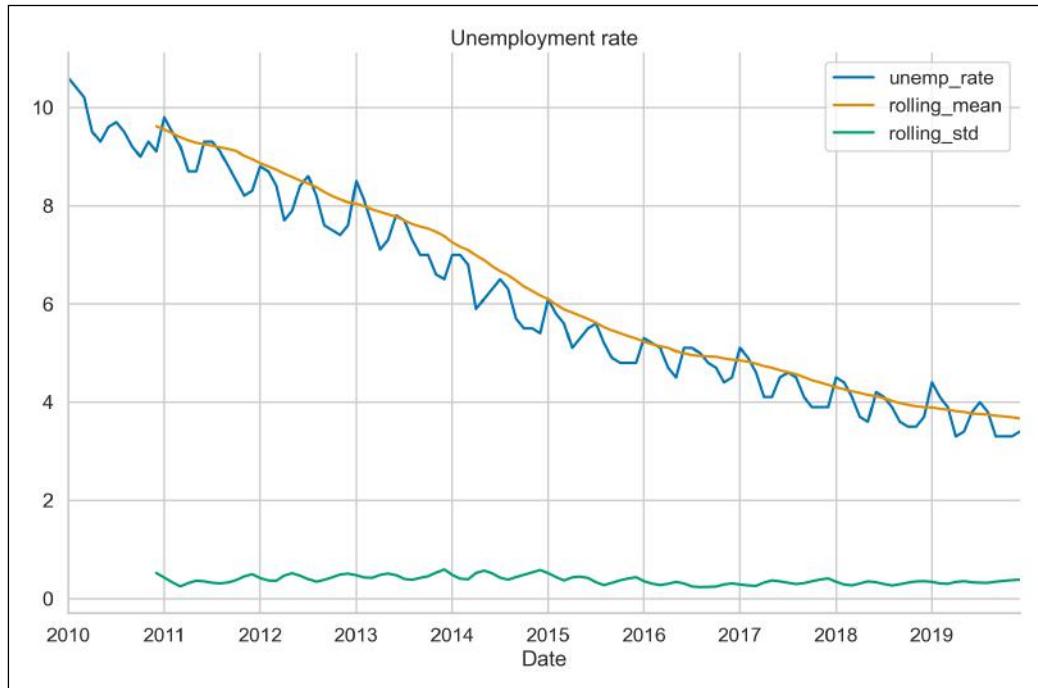


Figure 6.3: The US unemployment rate together with the rolling average and standard deviation

From the analysis of *Figure 6.3*, we can infer that the trend and seasonal components seem to have a linear pattern. Therefore, we will use additive decomposition in the next step.

4. Carry out the seasonal decomposition using the additive model:

```
decomposition_results = seasonal_decompose(df["unemp_rate"],
                                             model="additive")
(
    decomposition_results
    .plot()
    .suptitle("Additive Decomposition")
)
```

Running the snippet generates the following plot:

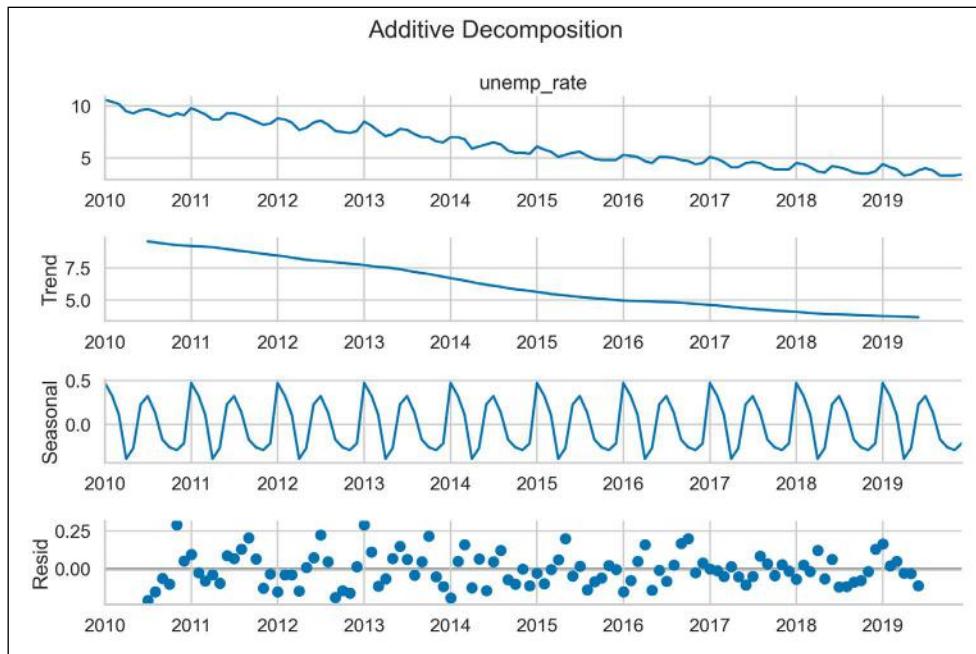


Figure 6.4: The seasonal decomposition of the US unemployment rate (using an additive model)

In the decomposition plot, we can see the extracted component series: trend, seasonal, and random (residual). To evaluate whether the decomposition makes sense, we can look at the random component. If there is no discernible pattern (in other words, the random component is indeed random and behaves consistently over time), then the fit makes sense. In this case, it looks like the variance in the residuals is slightly higher in the first half of the dataset. This can indicate that a constant seasonal pattern is not good enough to accurately capture the seasonal component of the analyzed time series.

How it works...

After downloading the data in *Step 2*, we used the `rolling` method of a pandas DataFrame to calculate the rolling statistics. We specified that we wanted to use the window size of 12 months, as we are working with monthly data.

We used the `seasonal_decompose` function from the `statsmodels` library to carry out the classical decomposition. When doing so, we indicated what kind of model we would like to use—the possible values are `additive` and `multiplicative`.



When using `seasonal_decompose` with an array of numbers, we must specify the frequency of the observations (the `freq` argument) unless we are working with a pandas Series object. If we have missing values or want to extrapolate the residuals for the missing periods at the beginning and the end of the time series, we can pass an extra argument `extrapolate_trend='freq'`.

There's more...

The seasonal decomposition we have used in this recipe is the most basic approach. It comes with a few disadvantages:

- As the algorithm uses centered moving averages to estimate the trend, running the decomposition results in missing values of the trend line (and the residuals) at the very beginning and end of the time series.
- The seasonal pattern estimated using this approach is assumed to repeat every year. It goes without saying that this is a very strong assumption, especially for longer time series.
- The trend line has a tendency to over-smooth the data, which in turn results in the trend line not responding adequately to sharp or sudden fluctuations.
- The method is not robust to potential outliers in the data.

Over time, a few alternative approaches to time series decomposition were introduced. In this section, we will also cover seasonal and trend decomposition using **LOESS** (STL decomposition), which is implemented in the `statsmodels` library.



LOESS stands for **locally estimated scatterplot smoothing** and it is a method of estimating non-linear relationships.

We will not go into the details of how STL decomposition works; however, it makes sense to be familiar with its advantages over the other approaches:

- STL can handle any kind of seasonality (not restricted to monthly or quarterly, as some other methods are)
- The user can control the smoothness of the trend
- The seasonal component can change over time (the rate of change can be controlled by the user)
- More robust to outliers—the estimation of the trend and seasonal components is not affected by their presence, while their impact is still visible in the remainder component

Naturally, it is not a silver-bullet solution and comes with some drawbacks of its own. For example, STL can only be used with additive decomposition and it does not automatically account for trading days/calendar variations.



There is a recent variant of STL decomposition that can handle multiple seasonalities. For example, a time series of hourly data can exhibit daily/weekly/monthly seasonalities. The approach is called **Multiple Seasonal-Trend Decomposition using LOESS (MSTL)** and you can find the reference to it in the *See also* section.

We can carry out the STL decomposition with the following snippet:

```
from statsmodels.tsa.seasonal import STL

stl_decomposition = STL(df[["unemp_rate"]]).fit()
stl_decomposition.plot() \
    .suptitle("STL Decomposition")
```

Running the code generates the following plot:

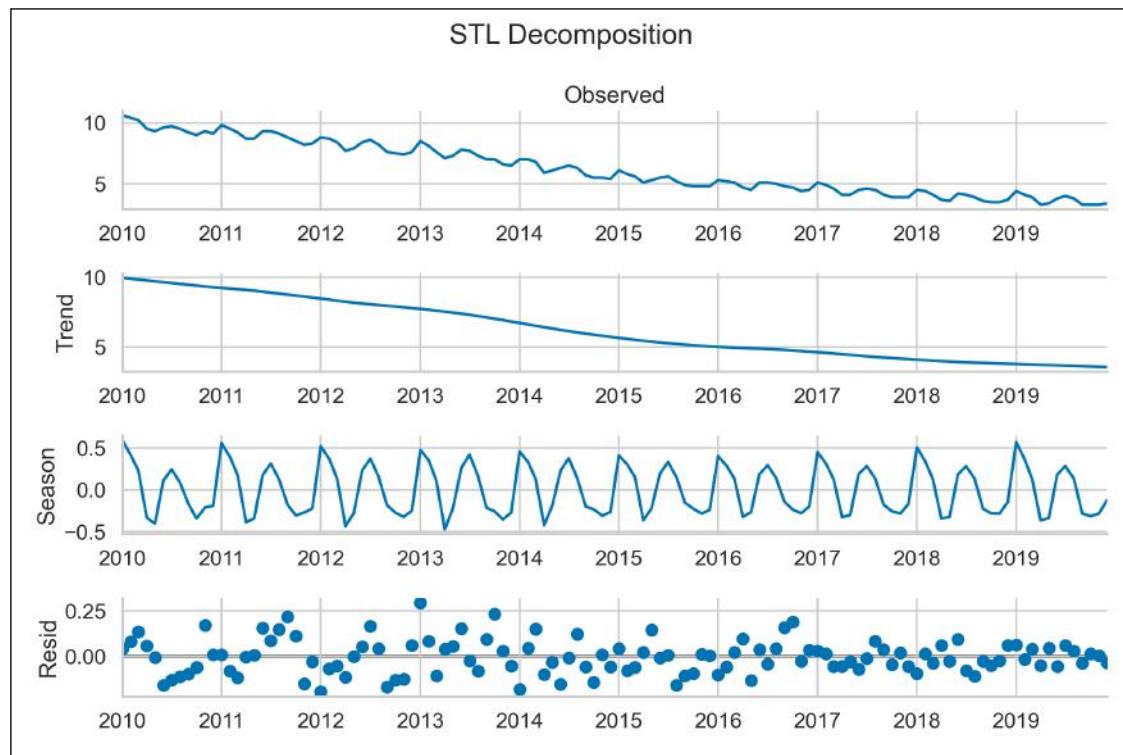


Figure 6.5: The STL decomposition of the US unemployment time series

We can see that the decomposition plots of the STL and classical decompositions are very similar. However, there are some nuances in *Figure 6.5*, connected to the benefits of STL decomposition over the classical one. First, there are no missing values in the trend estimate. Second, the seasonal component is slowly changing over time. You can see it clearly when looking at, for example, the values for January across the years.



The default value of the `seasonal` argument in `STL` is set to 7, but the authors of the approach suggest using larger values (must be odd integers greater than or equal to 7). Under the hood, the value of that parameter indicates the number of consecutive years to be used in estimating each value of the seasonal component. The larger the chosen value, the smoother the seasonal component becomes. This, in turn, causes fewer of the variations observed in the time series to be attributed to the seasonal component. The interpretation is similar for the `trend` argument, though it represents the number of consecutive observations to be used for estimating the trend component.

We have also mentioned that one of the benefits of the `STL` decomposition is its higher robustness against outliers. We can use the `robust` argument to switch on a data-dependent weighting function. It re-weights the observations when estimating the LOESS, which becomes **LOWESS** (locally weighted scatterplot smoothing) in such a scenario. When using robust estimation, the model can tolerate larger errors that are visible on the residual component's plot.

In *Figure 6.6* you can see a comparison of fitting two `STL` decompositions to the US unemployment data—with and without the robust estimation. For the code used to generate the figure, please refer to the notebook in the book's GitHub repository.

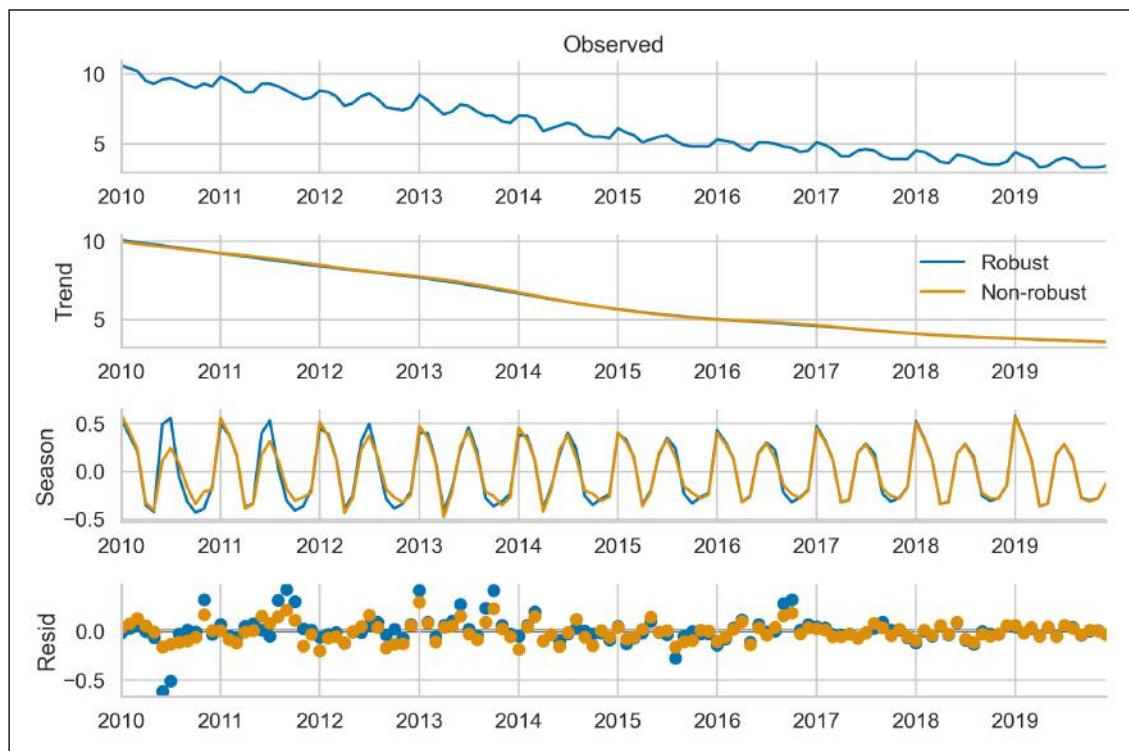


Figure 6.6: The effect of using robust estimation in the `STL` decomposition process



We can clearly observe the effects of using robust estimation—larger errors are tolerated and the shape of the seasonal component is different over the first few years of the analyzed time series. There is no clear answer whether the robust or non-robust approach is better in this case; it all depends on what we want to use the decomposition for. Seasonal decomposition methods presented in this recipe can also serve as simple outlier detection algorithms. For example, we could decompose the series, extract the residuals, and flag observations as outliers when their residuals are outside 3 times the **interquartile range (IQR)**. The `kats` library provides an implementation of such an algorithm in its `OutlierDetector` class.

Other available approaches to seasonal decomposition include:

- **Seasonal Extraction in ARIMA Time Series (SEATS)** decomposition.
- X11 decomposition—this variant of the decomposition creates a trend-cycle component for all observations and allows the seasonal component to change slowly over time.
- Hodrick-Prescott filter—while this method is not really a seasonal decomposition approach, it is a data smoothing technique used to remove short-term fluctuations associated with the business cycle. By removing those, we can reveal the long-term trends. The HP filter is commonly used in macroeconomics. You can find its implementation in the `hpfilter` function of `statsmodels`.

See also

Useful references on time series decomposition:

- Bandara, K., Hyndman, R. J., & Bergmeir, C. 2021. “MSTL: A Seasonal-Trend Decomposition Algorithm for Time Series with Multiple Seasonal Patterns.” *arXiv preprint arXiv:2107.13462*.
- Cleveland, R. B., Cleveland, W. S., McRae, J. E., & Terpenning, I. J. 1990. “A Seasonal Trend Decomposition Procedure Based on LOESS,” *Journal of Official Statistics* 6(1): 3–73.
- Hyndman, R.J. & Athanasopoulos, G. 2021. *Forecasting: Principles and Practice*, 3rd edition, OTexts: Melbourne, Australia. OTexts.com/fpp3
- Sutcliffe, A. 1993. *X11 time series decomposition and sampling errors*. Australian Bureau of Statistics.

Testing for stationarity in time series

One of the most important concepts in time series analysis is stationarity. Plainly speaking, a stationary time series is a series whose properties do not depend on the time at which the series is observed. In other words, stationarity implies that the statistical properties of the **data-generating process (DGP)** of a certain time series do not change over time.

Hence, we should not be able to see any trend or seasonal patterns in a stationary time series, as their existence violates the stationarity assumptions. On the other hand, a white noise process is stationary, as it does not matter when we observe it; it will always look pretty much the same at any point in time.



A time series without trend and seasonality but with cyclic behavior can still be stationary because the cycles are not of a fixed length. So unless we explicitly observe a time series, we cannot be sure where the peaks and troughs of the cycles will be located.

To put it more formally, there are multiple definitions of stationarity, some stricter in terms of the assumptions than others. For practical use cases, we can work with the one called **weak stationarity** (or covariance stationarity). For a time series to be classified as (covariance) stationary, it must satisfy the following three conditions:

- The mean of the series must be constant
- The variance of the series must be finite and constant
- The covariance between periods of identical distance must be constant

Stationarity is a desired characteristic of time series as it makes modeling and extrapolating (forecasting) into the future more feasible. That is because a stationary series is easier to predict than a non-stationary one, as its statistical properties will be the same in the future as they have been in the past.

Some drawbacks of non-stationary data are:

- Variance can be misspecified by the model
- Worse model fit, resulting in worse forecasts
- We cannot leverage valuable time-dependent patterns in the data



While stationarity is a desired trait of a time series, this is not applicable to all statistical models. We would want our time series to be stationary when modeling the series using some kind of auto-regressive model (AR, ARMA, ARIMA, and so on). However, there are also models that do not benefit from stationary time series, for example, those that depend heavily on time series decomposition (exponential smoothing methods or Facebook's Prophet).

In this recipe, we will show you how to test the time series for stationarity. To do so, we employ the following methods:

- The **Augmented Dickey-Fuller (ADF)** test
- The **Kwiatkowski-Phillips-Schmidt-Shin (KPSS)** test
- Plots of the (partial) autocorrelation function (PACF/ACF)

We will investigate the stationarity of monthly unemployment rates in the years 2010 to 2019.

Getting ready

We will use the same data that we used in the *Time series decomposition* recipe. In the plot presenting the rolling mean and standard deviation of the unemployment rates (*Figure 6.3*), we have already seen a negative trend over time, suggesting non-stationarity.

How to do it...

Execute the following steps to test if the time series of monthly US unemployment rates is stationary:

1. Import the libraries:

```
import pandas as pd
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller, kpss
```

2. Define a function for running the ADF test:

```
def adf_test(x):
    indices = ["Test Statistic", "p-value",
               "# of Lags Used", "# of Observations Used"]

    adf_test = adfuller(x, autolag="AIC")
    results = pd.Series(adf_test[0:4], index=indices)

    for key, value in adf_test[4].items():
        results[f"Critical Value ({key})"] = value

    return results
```

Having defined the function, we can run the test:

```
adf_test(df["unemp_rate"])
```

Running the snippet generates the following summary:

Test Statistic	-2.053411
p-value	0.263656
# of Lags Used	12.000000
# of Observations Used	107.000000
Critical Value (1%)	-3.492996
Critical Value (5%)	-2.888955
Critical Value (10%)	-2.581393

The null hypothesis of the ADF test states that the time series is not stationary. With a p-value of 0.26 (or equivalently, the test statistic is greater than the critical value for the selected confidence level), we have no reason to reject the null hypothesis, meaning that we can conclude that the series is not stationary.

3. Define a function for running the KPSS test:

```
def kpss_test(x, h0_type="c"):  
    indices = ["Test Statistic", "p-value", "# of Lags"]  
  
    kpss_test = kpss(x, regression=h0_type)  
    results = pd.Series(kpss_test[0:3], index=indices)  
  
    for key, value in kpss_test[3].items():  
        results[f"Critical Value ({key})"] = value  
  
    return results
```

Having defined the function, we can run the test:

```
kpss_test(df["unemp_rate"])
```

Running the snippet generates the following summary:

Test Statistic	1.799224
p-value	0.010000
# of Lags	6.000000
Critical Value (10%)	0.347000
Critical Value (5%)	0.463000
Critical Value (2.5%)	0.574000
Critical Value (1%)	0.739000

The null hypothesis of the KPSS test states that the time series is stationary. With a p-value of **0.01** (or a test statistic greater than the selected critical value), we have reasons to reject the null hypothesis in favor of the alternative one, indicating that the series is not stationary.

4. Generate the ACF/PACF plots:

```
N_LAGS = 40  
SIGNIFICANCE_LEVEL = 0.05  
  
fig, ax = plt.subplots(2, 1)  
plot_acf(df["unemp_rate"], ax=ax[0], lags=N_LAGS,  
         alpha=SIGNIFICANCE_LEVEL)  
plot_pacf(df["unemp_rate"], ax=ax[1], lags=N_LAGS,  
         alpha=SIGNIFICANCE_LEVEL)
```

Running the snippet generates the following plots:

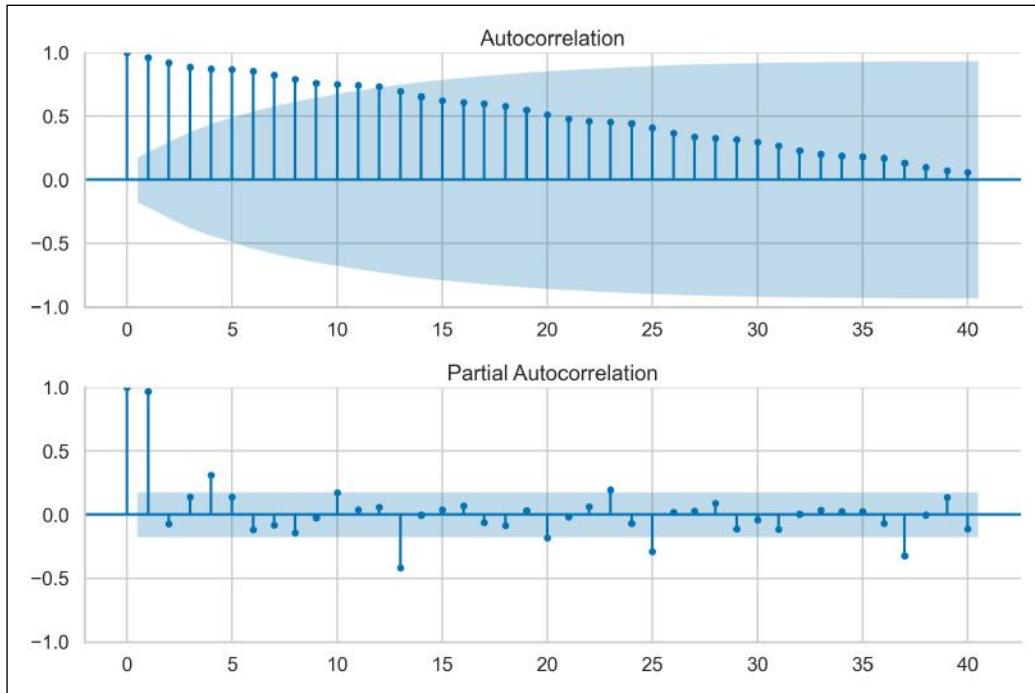


Figure 6.7: Autocorrelation and Partial Autocorrelation plots of the unemployment rate

In the ACF plot, we can see that there are significant autocorrelations (above the 95% confidence interval, corresponding to the selected 5% significance level). There are also some significant autocorrelations at lags 1 and 4 in the PACF plot.

How it works...

In *Step 2*, we defined a function used for running the ADF test and printing out the results. We specified `autolag="AIC"` while calling the `adfuller` function, so the number of considered lags is automatically selected based on the **Akaike Information Criterion (AIC)**. Alternatively, we could select the number of lags manually.

For the `kpss` function (*Step 3*), we specified the `regression` argument. A value of "c" corresponds to the null hypothesis stating that the series is level-stationary, while "ct" corresponds to trend-stationary (removing the trend from the series would make it level-stationary).

For all the tests and the autocorrelation plots, we selected a significance level of 5%, which indicates the probability of rejecting the null hypothesis (H_0) when it is, in fact, true.

There's more...

In this recipe, we have used the `statsmodels` library to carry out the stationarity tests. However, we had to wrap its functionalities in custom functions to have a nicely presented summary. Alternatively, we can use the stationarity tests from the `arch` library (we will cover the library in more depth when we explore the GARCH models in *Chapter 9, Modeling Volatility with GARCH Class Models*).

We can carry out the ADF test using the following snippet:

```
from arch.unitroot import ADF
adf = ADF(df["unemp_rate"])
print(adf.summary().as_text())
```

Which returns a nicely formatted output containing all the relevant information:

```
Augmented Dickey-Fuller Results
=====
Test Statistic          -2.053
P-value                 0.264
Lags                   12
-----
Trend: Constant
Critical Values: -3.49 (1%), -2.89 (5%), -2.58 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
```

The `arch` library also contains more stationarity tests, including:

- The Zivot-Andrews test (also available in `statsmodels`)
- The **Phillips-Perron (PP)** test (unavailable in `statsmodels`)

A potential drawback of the ADF and KPSS tests is that they do not allow for the possibility of a structural break, that is, an abrupt change in the mean or other parameters of the data-generating process. The Zivot-Andrews test allows for the possibility of a single structural break in the series, with an unknown time of its occurrence.

We can run the test using the following snippet:

```
from arch.unitroot import ZivotAndrews
za = ZivotAndrews(df["unemp_rate"])
print(za.summary().as_text())
```

Which generates the summary:

```
Zivot-Andrews Results
=====
Test Statistic           -2.551
P-value                 0.982
Lags                   12
-----
Trend: Constant
Critical Values: -5.28 (1%), -4.81 (5%), -4.57 (10%)
Null Hypothesis: The process contains a unit root with a single structural break.
Alternative Hypothesis: The process is trend and break stationary.
```

Based on the test's p-value, we cannot reject the null hypothesis stating that the process is not stationary.

See also

For more information on the additional stationarity tests, please refer to:

- Phillips, P. C. B. & P. Perron, 1988. "Testing for a unit root in time series regression," *Biometrika* 75: 335-346.
- Zivot, E. & Andrews, D.W.K., 1992. "Further evidence on the great crash, the oil-price shock, and the unit-root hypothesis," *Journal of Business & Economic Studies*, 10: 251-270.

Correcting for stationarity in time series

In the previous recipe, we learned how to investigate if a given time series is stationary. In this one, we will investigate how to make a non-stationary time series stationary by using one (or multiple) of the following transformations:

- Deflation—accounting for inflation in monetary series using the **Consumer Price Index (CPI)**
- Applying the natural logarithm—making the potential exponential trend closer to linear and reducing the variance of the time series
- Differencing—taking the difference between the current observation and a lagged value (observation x time points before the current observation)

For this exercise, we will use monthly gold prices from the years 2000 to 2010. We have chosen this sample on purpose, as over that period the price of gold exhibits a consistently increasing trend—the series is definitely not stationary.

How to do it...

Execute the following steps to transform the series from non-stationary to stationary:

1. Import the libraries and authenticate and update the inflation data:

```
import pandas as pd
import numpy as np
import nasdaqdatalink
import cpi
from datetime import date
from chapter_6_utils import test_autocorrelation

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

In this recipe, we will be using the `test_autocorrelation` helper function, which combines the components we covered in the previous recipe, the ADF and KPSS tests, together with the ACF/PACF plots.

2. Download the prices of gold and resample to monthly values:

```
df = (
    nasdaqdatalink.get(dataset="WGC/GOLD_MONAVG_USD",
                        start_date="2000-01-01",
                        end_date="2010-12-31")
    .rename(columns={"Value": "price"})
    .resample("M")
    .last()
)
```

We can use the `test_autocorrelation` helper function to test if the series is stationary. We have done so in the notebook (available on GitHub) and the time series of monthly gold prices is indeed not stationary.

3. Deflate the gold prices (to the 2010-12-31 USD values) and plot the results:

```
DEFL_DATE = date(2010, 12, 31)

df["dt_index"] = pd.to_datetime(df.index)
df["price_deflated"] = df.apply(
    lambda x: cpi.inflate(x["price"], x["dt_index"], DEFL_DATE),
    axis=1
)
(
    df.loc[:, ["price", "price_deflated"]]
    .plot(title="Gold Price (deflated)")
)
```

Running the snippet generates the following plot:

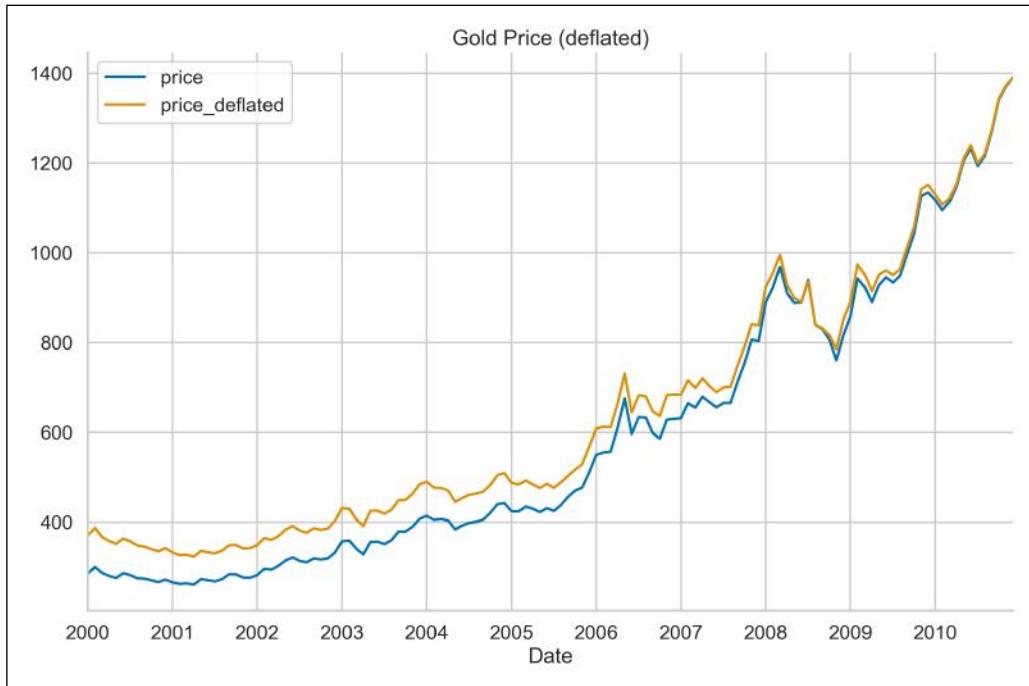


Figure 6.8: Monthly gold prices and the deflated time series

We could also adjust the gold prices to another point in time, as long as it is the same point for the entire series.

4. Apply the natural logarithm to the deflated series and plot it together with the rolling metrics:

```
WINDOW = 12
selected_columns = ["price_log", "rolling_mean_log",
                    "rolling_std_log"]

df["price_log"] = np.log(df.price_deflated)
df["rolling_mean_log"] = df.price_log.rolling(WINDOW) \
                        .mean()
df["rolling_std_log"] = df.price_log.rolling(WINDOW) \
                        .std()

(
    df[selected_columns]
    .plot(title="Gold Price (deflated + logged)",
          subplots=True)
)
```

Running the snippet generates the following plot:

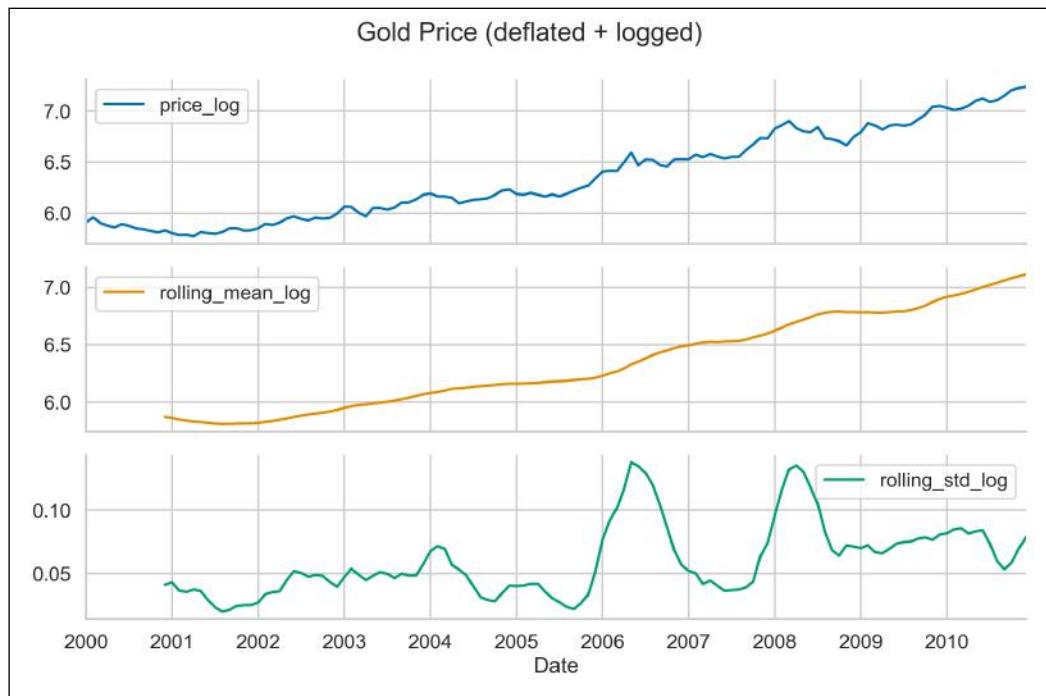


Figure 6.9: Time series after applying the deflation and natural logarithm, together with its rolling statistics

From the preceding plot, we can see that the log transformation did its job, that is, it made the exponential trend linear.

5. Use the `test_autocorrelation` (helper function for this chapter) to investigate if the series became stationary:

```
fig = test_autocorrelation(df["price_log"])
```

Running the snippet generates the following plot:

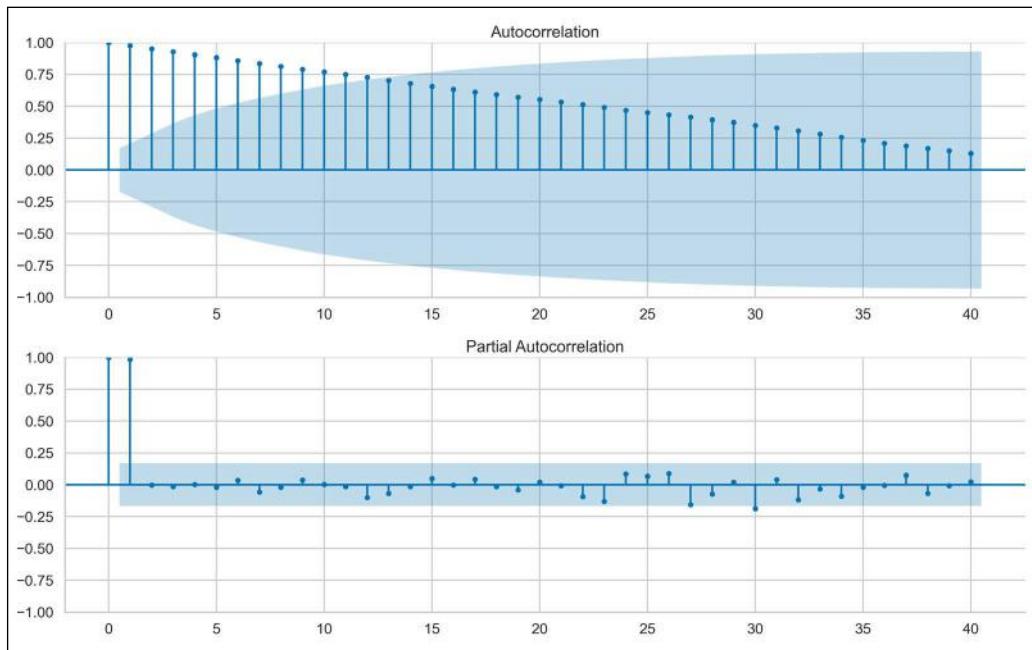


Figure 6.10: The ACF and PACF plots of the transformed time series

We also print the results of the statistical tests:

```
ADF test statistic: 1.04 (p-val: 0.99)
KPSS test statistic: 1.93 (p-val: 0.01)
```

After inspecting the results of the statistical tests and the ACF/PACF plots, we can conclude that deflation and a natural algorithm were not enough to make the time series of monthly gold prices stationary.

6. Apply differencing to the series and plot the results:

```
selected_columns = ["price_log_diff", "roll_mean_log_diff",
                    "roll_std_log_diff"]

df["price_log_diff"] = df.price_log.diff(1)
df["roll_mean_log_diff"] = df.price_log_diff.rolling(WINDOW) \
                           .mean()
```

```
df["roll_std_log_diff"] = df.price_log_diff.rolling(WINDOW) \
    .std()
df[selected_columns].plot(title="Gold Price (deflated + log + diff)")
```

Running the snippet generates the following plot:

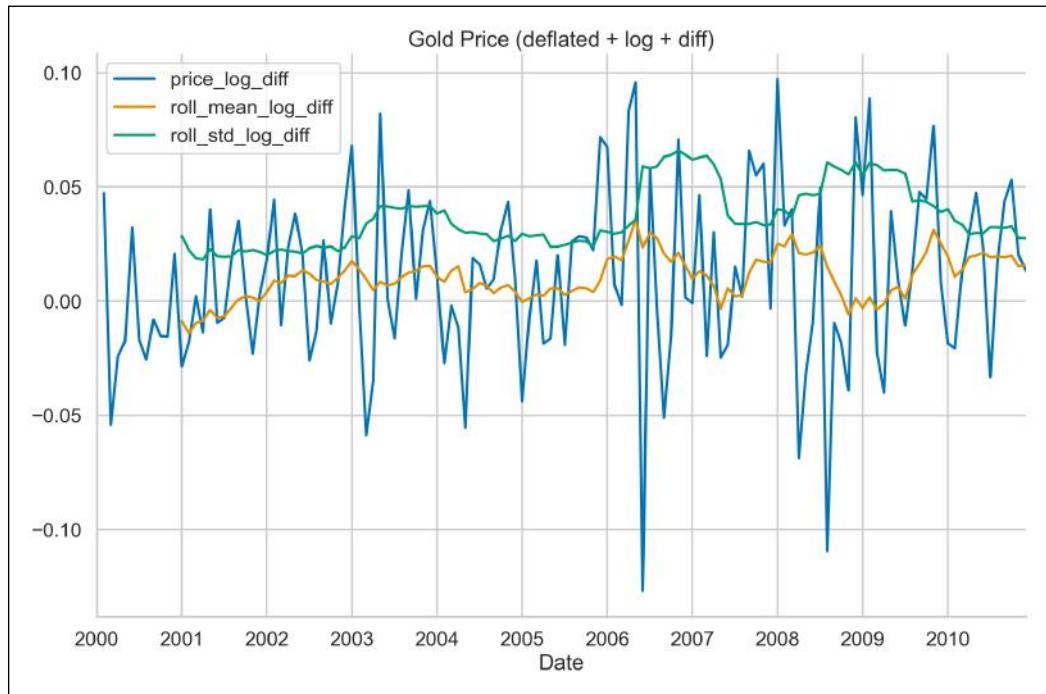


Figure 6.11: Time series after applying three types of transformations, together with its rolling statistics

The transformed gold prices give the impression of being stationary—the series oscillates around 0 with no visible trend and approximately constant variance.

7. Test if the series became stationary:

```
fig = test_autocorrelation(df["price_log_diff"].dropna())
```

Running the snippet generates the following plot:

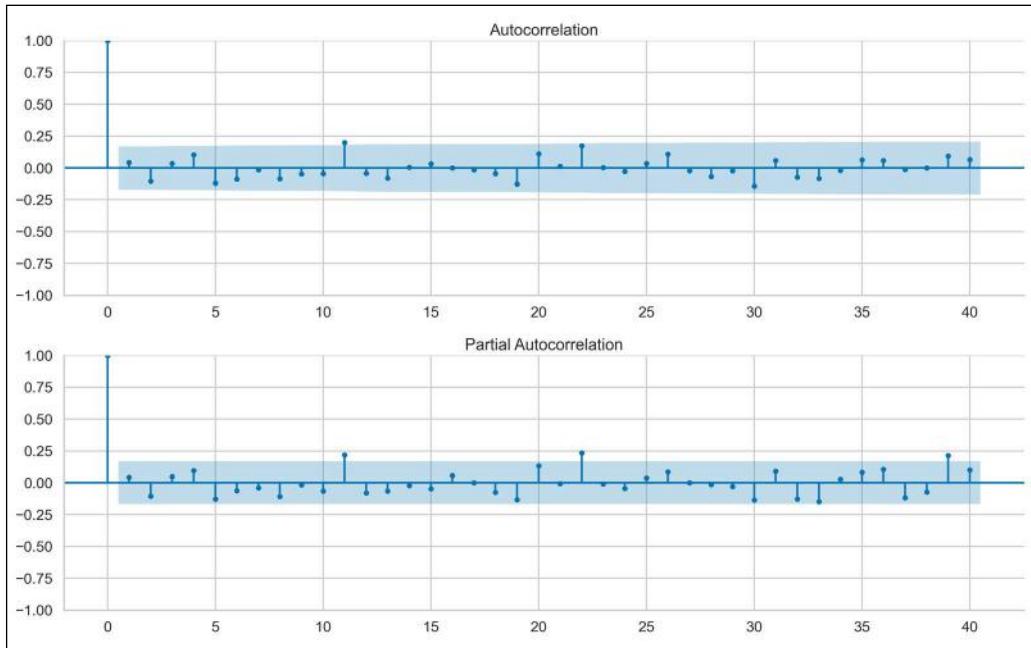


Figure 6.12: The ACF and PACF plots of the transformed time series

We also print the results of the statistical tests:

```
ADF test statistic: -10.87 (p-val: 0.00)
KPSS test statistic: 0.30 (p-val: 0.10)
```

After applying the first differences, the series became stationary at the 5% significance level (according to both tests). In the ACF/PACF plots, we can see that there were a few significant values of the function at lags 11, 22, and 39. This might indicate some kind of seasonality or simply be a false signal. Using a 5% significance level means that 5% of the values might lie outside the 95% confidence interval—even when the underlying process does not show any autocorrelation or partial autocorrelation.

How it works...

After importing the libraries, authenticating, and potentially updating the CPI data, we downloaded the monthly gold prices from Nasdaq Data Link. There were some duplicate values in the series. For example, there were entries for 2000-04-28 and 2000-04-30, both with the same value. To deal with this issue, we resampled the data to monthly frequency by taking the last available value.

By doing so, we only removed potential duplicates in each month, without changing any of the actual values. In Step 3, we used the `cpi` library to deflate the time series by accounting for inflation in the US dollar. The library relies on the CPI-U index recommended by the Bureau of Labor Statistics. To make it work, we created an artificial index column containing dates as objects of the `datetime.date` class. The `inflate` function takes the following arguments:

- `value`—the dollar value we want to adjust.
- `year_or_month`—the date that the dollar value comes from.
- `to`—optionally, the date we want to adjust to. If we don't provide this argument, the function will adjust to the most recent year.

In *Step 4*, we applied the natural logarithm (`np.log`) to all the values to transform what looked like an exponential trend into linear. This operation was applied to prices that had already been corrected for inflation.

As the last transformation, we used the `diff` method of a pandas DataFrame to calculate the difference between the value in time t and time $t-1$ (the default setting corresponds to the first difference). We can specify a different number by changing the `period` argument.

There's more...

The considered gold prices do not contain obvious seasonality. However, if the dataset shows seasonal patterns, there are a few potential solutions:

- Adjustment by differencing—instead of using first-order differencing, use a higher-order one, for example, if there is yearly seasonality in monthly data, use `diff(12)`.
- Adjustment by modeling—we can directly model the seasonality and then remove it from the series. One possibility is to extract the seasonal component from the `seasonal_decompose` function or another more advanced automatic decomposition algorithm. In this case, we should subtract the seasonal component when using the additive model or divide by it if the model is multiplicative. Another solution would be to use `np.polyfit()` to fit the best polynomial of a chosen order to the selected time series and then subtract it from the original series.

The **Box-Cox transformation** is another type of adjustment we can use on the time series data. It combines different exponential transformation functions to make the distribution more similar to the Normal (Gaussian) distribution. We can use the `boxcox` function from the `scipy` library, which allows us to automatically find the value of the `lambda` parameter for the best fit. One condition to be aware of is that all the values in the series must be positive, so the transformation should not be used after calculating the first differences or any other transformations that potentially introduce negative values to the series.

A library called `pmdarima` (more on this library can be found in the following recipes) contains two functions that employ statistical tests to determine how many times we should differentiate the series in order to achieve stationarity (and also remove seasonality, that is, seasonal stationarity).

We can employ the following tests to investigate stationarity: ADF, KPSS, and Phillips-Perron:

```
from pmdarima.arima import ndiffs, nsdiffs

print(f"Suggested # of differences (ADF): {ndiffs(df['price'], test='adf')}")
print(f"Suggested # of differences (KPSS): {ndiffs(df['price'], test='kpss')}")
print(f"Suggested # of differences (PP): {ndiffs(df['price'], test='pp')}")
```

Running the snippet returns the following:

```
Suggested # of differences (ADF): 1
Suggested # of differences (KPSS): 2
Suggested # of differences (PP): 1
```

For the KPSS test, we can also specify what type of null hypothesis we want to test against. The default is level stationarity (`null="level"`). The results of the tests, or more precisely the need for differencing, suggest that the series without any differencing is not stationary.

The library also contains two tests for seasonal differences:

- **Osborn, Chui, Smith, and Birchenhall** (OCSB)
- **Canova-Hansen** (CH)

To run them, we also need to specify the frequency of our data. In our case, it is 12, as we are working with monthly data:

```
print(f"Suggested # of differences (OCSB): {nsdiffs(df['price'], m=12,
test='ocsb')}")

print(f"Suggested # of differences (CH): {nsdiffs(df['price'], m=12,
test='ch')}")
```

The output is as follows:

```
Suggested # of differences (OCSB): 0
Suggested # of differences (CH): 0
```

The results suggest no seasonality in gold prices.

Modeling time series with exponential smoothing methods

Exponential smoothing methods are one of the two families of classical forecasting models. Their underlying idea is that forecasts are simply weighted averages of past observations. When calculating those averages, more emphasis is put on recent observations. To achieve that, the weights are decaying exponentially with time. These models are suitable for non-stationary data, that is, data with a trend and/or seasonality. Smoothing methods are popular because they are fast (not a lot of computations are required) and relatively reliable when it comes to forecasts' accuracy.

Collectively, the exponential smoothing methods can be defined in terms of the **ETS framework (Error, Trend, and Season)**, as they combine the underlying components in the smoothing calculations. As in the case of the seasonal decomposition, those terms can be combined additively, multiplicatively, or simply left out of the model.



Please see *Forecasting: Principles and Practice* (Hyndman and Athanasopoulos) for more information on the taxonomy of exponential smoothing methods.

The simplest model is called **simple exponential smoothing (SES)**. This class of models is most apt for cases when the considered time series does not exhibit any trend or seasonality. They also work well with series with only a few data points.

The model is parameterized by a smoothing parameter α with values between 0 and 1. The higher the value, the more weight is put on recent observations. When $\alpha = 0$, the forecasts for the future are equal to the average of training data. When $\alpha = 1$, all the forecasts have the same value as the last observation in the training set.

The forecasts produced using SES are flat, that is, regardless of the time horizon, all forecasts have the same value (corresponding to the last level component). That is why this method is only suitable for series with neither trend nor seasonality.

Holt's linear trend method (also known as Holt's double exponential smoothing method) is an extension of SES that accounts for a trend in the series by adding the trend component to the model's specification. As a consequence, this model should be used when there is a trend in the data, but it still cannot handle seasonality.

One issue with Holt's model is that the trend is constant in the future, which means that it increases/decreases indefinitely. That is why an extension of the model dampens the trend by adding the dampening parameter, φ . It makes the trend converge to a constant value in the future, effectively flattening it.



φ is rarely smaller than 0.8, as the dampening has a very strong effect for smaller values of φ . The best practice is to restrict the values of φ so that they lie between 0.8 and 0.98. For $\varphi = 1$ the damped model is equivalent to the model without dampening.

Lastly, we will cover the extension of Holt's method called **Holt-Winters' seasonal smoothing** (also known as Holt-Winters' triple exponential smoothing). As the name suggests, it accounts for the seasonality in time series. Without going into too much detail, this method is most suitable for data with both trend and seasonality.

There are two variants of this model and they have either additive or multiplicative seasonalities. In the former one, the seasonal variations are more or less constant throughout the time series. In the latter one, the variations change in proportion to the passing of time.

In this recipe, we will show you how to apply the covered smoothing methods to monthly US unemployment rates (non-stationary data with trend and seasonality). We will fit the model to the prices from 2010 to 2018 and make forecasts for 2019.

Getting ready

We will use the same data that we used in the *Time series decomposition* recipe.

How to do it...

Execute the following steps to create forecasts of the US unemployment rate using the exponential smoothing methods:

- ### 1. Import the libraries:

```
import pandas as pd
from datetime import date
from statsmodels.tsa.holtwinters import (ExponentialSmoothing,
                                           SimpleExpSmoothing,
                                           Holt)
```

- ## 2. Create the train/test split:

```
TEST_LENGTH = 12
df.index.freq = "MS"
df_train = df.iloc[:-TEST_LENGTH]
df_test = df[-TEST_LENGTH:]
```

3. Fit two SES models and calculate the forecasts:

```
ses_1 = SimpleExpSmoothing(df_train).fit(smoothing_level=0.5)
ses_forecast_1 = ses_1.forecast(TEST_LENGTH)

ses_2 = SimpleExpSmoothing(df_train).fit()
ses_forecast_2 = ses_2.forecast(TEST_LENGTH)

ses_1.params.formatted
```

Running the snippet generates the following table:

	name	param	optimized
smoothing_level	alpha	0.500000	False
initial_level	l.0	10.358112	True

Figure 6.13: The values of the fitted coefficients for the first SES model

We can use the `summary` method to print a more detailed summary of the fitted model.

4. Combine the forecasts with the fitted values and plot them:

```

labels = [
    "unemp_rate",
    r"$\alpha=0.2$",
    r'$\alpha={0:.2f}$'.format(opt_alpha),
]
ax.legend(labels)

```

Running the snippet generates the following plot:

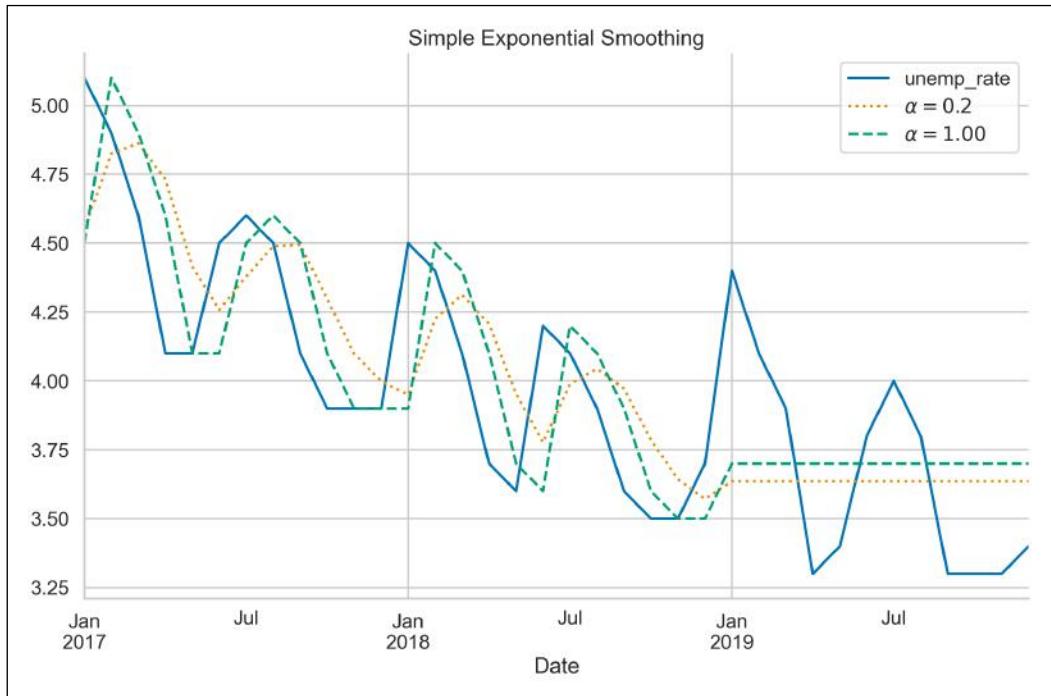


Figure 6.14: Modeling time series using SES

In *Figure 6.14*, we can observe the characteristic of SES that we described in the introduction to this recipe—the forecast is a flat line. We can also see that the optimal value that was selected by the optimization routine is equal to 1. Immediately, we can see the consequences of picking such a value: the fitted line of the model is effectively the line of the observed prices shifted to the right and the forecast is simply the last observed value.

5. Fit three variants of Holt's linear trend models and calculate the forecasts:

```

# Holt's model with Linear trend
hs_1 = Holt(df_train).fit()
hs_forecast_1 = hs_1.forecast(TEST_LENGTH)

# Holt's model with exponential trend
hs_2 = Holt(df_train, exponential=True).fit()

```

```

hs_forecast_2 = hs_2.forecast(TEST_LENGTH)

# Holt's model with exponential trend and damping
hs_3 = Holt(df_train, exponential=False,
            damped_trend=True).fit()
hs_forecast_3 = hs_3.forecast(TEST_LENGTH)

```

6. Plot the original series together with the models' forecasts:

```

hs_df = df.copy()
hs_df["hs_1"] = hs_1.fittedvalues.append(hs_forecast_1)
hs_df["hs_2"] = hs_2.fittedvalues.append(hs_forecast_2)
hs_df["hs_3"] = hs_3.fittedvalues.append(hs_forecast_3)

fig, ax = plt.subplots()
hs_df["2017":].plot(style=["-", ":" , "--", "-."], ax=ax,
                     title="Holt's Double Exponential Smoothing")
labels = [
    "unemp_rate",
    "Linear trend",
    "Exponential trend",
    "Exponential trend (damped)",
]
ax.legend(labels)

```

Running the snippet generates the following plot:

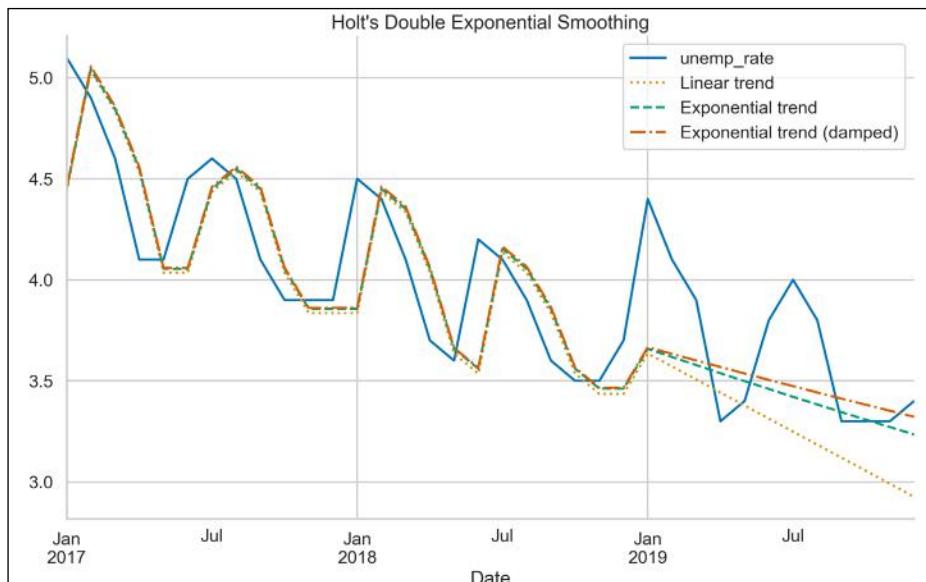


Figure 6.15: Modeling time series using Holt's Double Exponential Smoothing

We can already observe an improvement as, compared to the SES forecast, the lines are not flat anymore.

One additional thing worth mentioning is that while we were optimizing a single parameter `alpha (smoothing_level)` in the case of the SES, here we are also optimizing `beta (smoothing_trend)` and potentially also `phi (damping_trend)`.

7. Fit two variants of Holt-Winters' Triple Exponential Smoothing models and calculate the forecasts:

```
SEASONAL_PERIODS = 12

# Holt-Winters' model with exponential trend
hw_1 = ExponentialSmoothing(df_train,
                             trend="mul",
                             seasonal="add",
                             seasonal_periods=SEASONAL_PERIODS).fit()
hw_forecast_1 = hw_1.forecast(TEST_LENGTH)

# Holt-Winters' model with exponential trend and damping
hw_2 = ExponentialSmoothing(df_train,
                             trend="mul",
                             seasonal="add",
                             seasonal_periods=SEASONAL_PERIODS,
                             damped_trend=True).fit()
hw_forecast_2 = hw_2.forecast(TEST_LENGTH)
```

8. Plot the original series together with the models' results:

```
hw_df = df.copy()
hw_df["hw_1"] = hw_1.fittedvalues.append(hw_forecast_1)
hw_df["hw_2"] = hw_2.fittedvalues.append(hw_forecast_2)

fig, ax = plt.subplots()
hw_df["2017":].plot(
    style=["-", ":"], ax=ax,
    title="Holt-Winters' Triple Exponential Smoothing")
phi = hw_2.model.params["damping_trend"]

labels = [
    "unemp_rate",
    "Seasonal Smoothing",
    f"Seasonal Smoothing (damped with $\phi={phi:.2f}$)"]
]
ax.legend(labels)
```

Running the snippet generates the following plot:

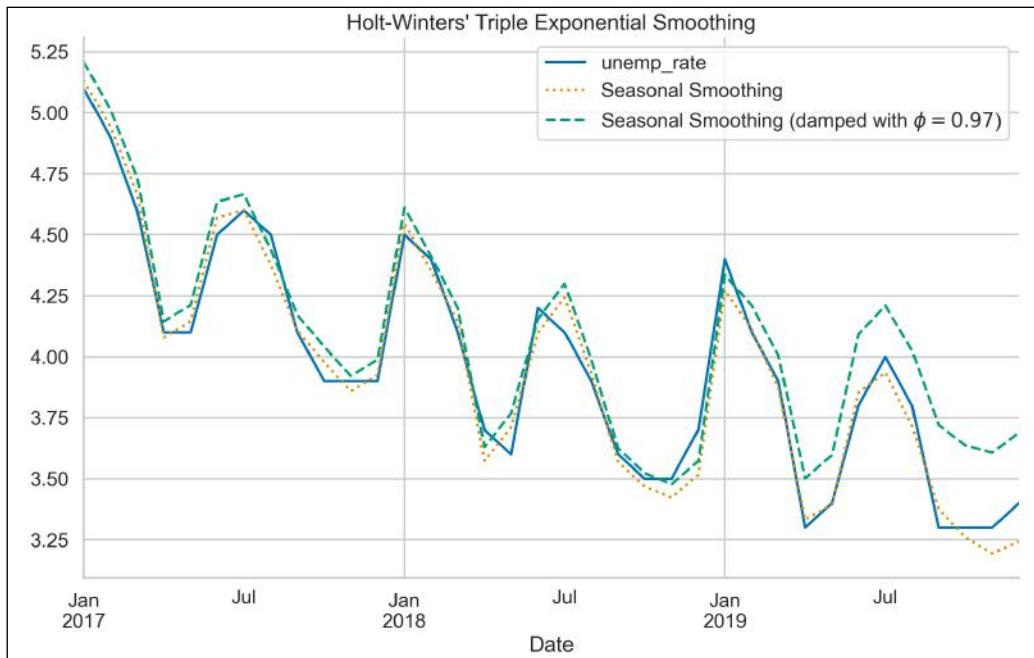


Figure 6.16: Modeling time series using Holt-Winters' Triple Exponential Smoothing

In the preceding plot, we can see that now the seasonal patterns were also incorporated into the forecasts.

How it works...

After importing the libraries, we fitted two different SES models using the `SimpleExpSmoothing` class and its `fit` method. To fit the model, we only used the training data. We could have manually selected the value of the smoothing parameter (`smoothing_level`), however, the best practice is to let `statsmodels` optimize it for the best fit. This optimization is done by minimizing the sum of squared residuals (errors). We created the forecasts using the `forecast` method, which requires the number of periods we want to forecast for (which, in our case, is equal to the length of the test set).

In *Step 3*, we combined the fitted values (accessed using the `fittedvalues` attribute of the fitted model) and the forecasts inside of a pandas DataFrame, together with the observed unemployment rate. We then visualized all the series. To make the plot easier to read, we capped the data to cover the last 2 years of the training set and the test set.

In *Step 5*, we used the `Holt` class (which is a wrapper around the more general `ExponentialSmoothing` class) to fit Holt's linear trend model. By default, the trend in the model is linear, but we can make it exponential by specifying `exponential=True` and adding dampening with `damped_trend=True`. As in the case of SES, using the `fit` method with no arguments results in running the optimization routine to determine the optimal values of the parameters. In *Step 6*, we again placed all the fitted values and forecasts into a DataFrame and then we visualized the results.

In *Step 7*, we estimated two variants of Holt-Winters' Triple Exponential Smoothing models. There is no separate class for this model, but we can adjust the `ExponentialSmoothing` class by adding the `seasonal` and `seasonal_periods` arguments. Following the taxonomy of the ETS models, we should indicate that the models have an additive seasonal component. In *Step 8*, we again put all the fitted values and forecasts into a `DataFrame` and then we visualized the results as a line plot.



When creating an instance of the `ExponentialSmoothing` class, we can additionally pass in the `use_boxcox` argument to automatically apply the Box-Cox transformation to the analyzed time series. Alternatively, we could use the log transformation by passing the "log" string to the same argument.

There's more...

In this recipe, we have fitted various exponential smoothing models to forecast the monthly unemployment rate. Each time, we specified what kind of model we were interested in and most of the time, we let `statsmodels` find the best-fitting parameters.

However, we could approach the task differently, that is, using a procedure called AutoETS. Without going into much detail, the goal of the procedure is to find the best-fitting flavor of an ETS model, given some constraints we provide upfront. You can read more about how the AutoETS procedure works in the references mentioned in the *See also* section.

The AutoETS procedure is available in the `sktime` library, which is a library/framework inspired by `scikit-learn`, but with a focus on time series analysis/forecasting.

Execute the following steps to find the best ETS model using the AutoETS approach:

1. Import the libraries:

```
from sktime.forecasting.ets import AutoETS  
from sklearn.metrics import mean_absolute_percentage_error
```

2. Fit the AutoETS model:

```
auto_ets = AutoETS(auto=True, n_jobs=-1, sp=12)  
auto_ets.fit(df_train.to_period())  
auto_ets_fcst = auto_ets.predict(fh=list(range(1, 13)))
```

3. Add the model's forecast to the plot of the Holt-Winters' forecasts:

```
auto_ets_df = hw_df.to_period().copy()  
auto_ets_df["auto_ets"] = (  
    auto_ets  
    ._fitted_forecaster  
    .fittedvalues  
    .append(auto_ets_fcst["unemp_rate"])  
)
```

```

fig, ax = plt.subplots()
auto_ets_df["2017"].plot(
    style=["-", ":" ,"--", "-."], ax=ax,
    title="Holt-Winters' models vs. AutoETS"
)
labels = [
    "unemp_rate",
    "Seasonal Smoothing",
    f"Seasonal Smoothing (damped with $\phi={phi:.2f}$)",
    "AutoETS",
]
ax.legend(labels)

```

Running the snippet generates the following plot:

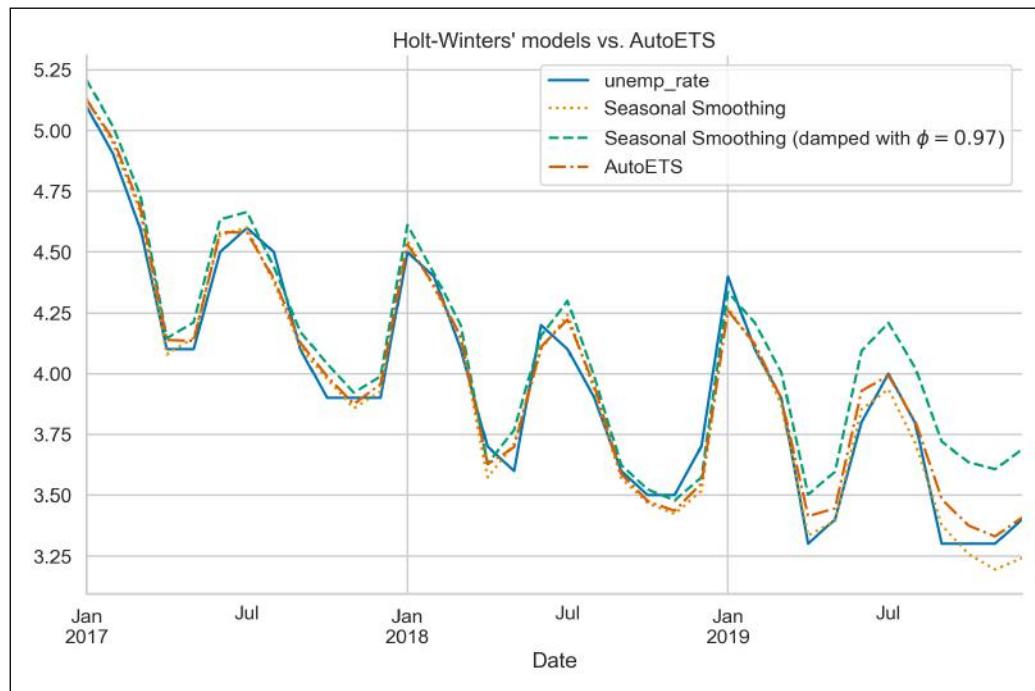


Figure 6.17: The results of the AutoETS forecast plotted over the results of the Holt-Winters' approach

In *Figure 6.17*, we can see that the in-sample fits of the Holt-Winters' model and AutoETS are very similar. When it comes to the forecast, they do differ and it is hard to say which one better predicts the unemployment rate.

That is why in the next step we calculate the **Mean Absolute Percentage Error (MAPE)**, which is a popular evaluation metric used in time series forecasting (and other fields).

4. Calculate the MAPEs of the Holt-Winters' forecasts and of AutoETS:

```
fcst_dict = {
    "Seasonal Smoothing": hw_forecast_1,
    "Seasonal Smoothing (damped)": hw_forecast_2,
    "AutoETS": auto_ets_fcst,
}

print("MAPEs ----")
for key, value in fcst_dict.items():
    mape = mean_absolute_percentage_error(df_test, value)
    print(f"{key}: {100 * mape:.2f}%")
```

Running the snippet generates the following summary:

```
MAPEs ----
Seasonal Smoothing: 1.81%
Seasonal Smoothing (damped): 6.53%
AutoETS: 1.78%
```

We can see that the accuracy scores (measured by MAPE) of the Holt-Winters' method and the AutoETS approach are very similar.

See also

Please see the following references for more information about the ETS methods:

- Hyndman, R. J., Akram, Md., & Archibald, 2008. “The admissible parameter space for exponential smoothing models,” *Annals of Statistical Mathematics*, **60** (2): 407–426.
- Hyndman, R. J., Koehler, A.B., Snyder, R.D., & Grose, S., 2002. “A state space framework for automatic forecasting using exponential smoothing methods,” *International J. Forecasting*, **18**(3): 439–454.
- Hyndman, R. J & Koehler, A. B., 2006. “Another look at measures of forecast accuracy,” *International Journal of Forecasting*, **22**(4): 679-688
- Hyndman, R. J., Koehler, A.B., Ord, J.K., & Snyder, R.D. 2008. *Forecasting with Exponential Smoothing: The State Space Approach*, Springer-Verlag. <http://www.exponentialsmoothing.net>.
- Hyndman, R. J. & Athanasopoulos, G. 2021. *Forecasting: Principles and Practice*, 3rd edition, OTexts: Melbourne, Australia. OTexts.com/fpp3.
- Winters, P.R. 1960. “Forecasting sales by exponentially weighted moving averages,” *Management Science* **6**(3): 324–342.

Modeling time series with ARIMA class models

ARIMA models are a class of statistical models that are used for analyzing and forecasting time series data. They aim to do so by describing the autocorrelations in the data. ARIMA stands for Autoregressive Integrated Moving Average and is an extension of a simpler ARMA model. The goal of the additional integration component is to ensure the stationarity of the series. That is because, in contrast to the exponential smoothing models, the ARIMA models require the time series to be stationary. Below we briefly go over the models' building blocks.

AR (autoregressive) model:

- This kind of model uses the relationship between an observation and its p lagged values
- In the financial context, the autoregressive model tries to account for the momentum and mean reversion effects

I (integration):

- Integration, in this case, refers to differencing the original time series (subtracting the value from the previous period from the current period's value) to make it stationary
- The parameter responsible for integration is d (called degree/order of differencing) and indicates the number of times we need to apply differencing

MA (moving average) model:

- This kind of model uses the relationship between an observation and the white noise terms (shocks that occurred in the last q observations).
- In the financial context, the moving average models try to account for the unpredictable shocks (observed in the residuals) that influence the observed time series. Some examples of such shocks could be natural disasters, breaking news connected to a certain company, etc.
- The white noise terms in the MA model are unobservable. Because of that, we cannot fit an ARIMA model using **ordinary least squares (OLS)**. Instead, we have to use an iterative estimation method such as **MLE (Maximum Likelihood Estimation)**.

All of these components fit together and are directly specified in the commonly used notation: ARIMA (p, d, q). In general, we should try to keep the values of the ARIMA parameters as small as possible in order to avoid unnecessary complexity and prevent overfitting to the training data. One possible rule of thumb would be to keep $d \leq 2$, while p and q should not be higher than 5. Also, most likely one of the terms (AR or MA) will dominate in the model, leading to the other one having a comparatively small value of the parameter.



ARIMA models are very flexible and by appropriately setting their hyperparameters, we can obtain some special cases:

- ARIMA (0,0,0): White noise
- ARIMA (0,1,0) without constant: Random walk
- ARIMA (p,0,q): ARMA(p, q)
- ARIMA (p,0,0): AR(p) model
- ARIMA (0,0,q): MA(q) model
- ARIMA (0,1,2): Damped Holt's model
- ARIMA (0,1,1) without constant: SES model
- ARIMA (0,2,2): Holt's linear method with additive errors

ARIMA models are still very popular in the industry as they deliver near state-of-the-art performance (mostly for short-horizon forecasts), especially when we are dealing with small datasets. In such cases, more advanced machine and deep learning models are not able to show their true power.

One of the known weaknesses of the ARIMA models in the financial context is their inability to capture volatility clustering, which is observed in most financial assets.

In this recipe, we will go through all the necessary steps to correctly estimate an ARIMA model and learn how to verify that it is a proper fit for the data. For this example, we will once again use the monthly US unemployment rate from the years 2010 to 2019.

Getting ready

We will use the same data that we used in the *Time series decomposition* recipe.

How to do it...

Execute the following steps to create forecasts of the US unemployment rate using ARIMA models:

1. Import the libraries:

```
import pandas as pd
import numpy as np
from statsmodels.tsa.arima.model import ARIMA
from chapter_6_utils import test_autocorrelation
from sklearn.metrics import mean_absolute_percentage_error
```

2. Create the train/test split:

```
TEST_LENGTH = 12
df_train = df.iloc[:-TEST_LENGTH]
df_test = df.iloc[-TEST_LENGTH:]
```

We create the train/test split just as we have done in the previous recipe. This way, we will be able to compare the performance of the two types of models.

3. Apply the log transformation and calculate the first differences:

```
df_train["unemp_rate_log"] = np.log(df_train["unemp_rate"])
df_train["first_diff"] = df_train["unemp_rate_log"].diff()

df_train.plot(subplots=True,
              title="Original vs transformed series")
```

Running the snippet generates the following figure:

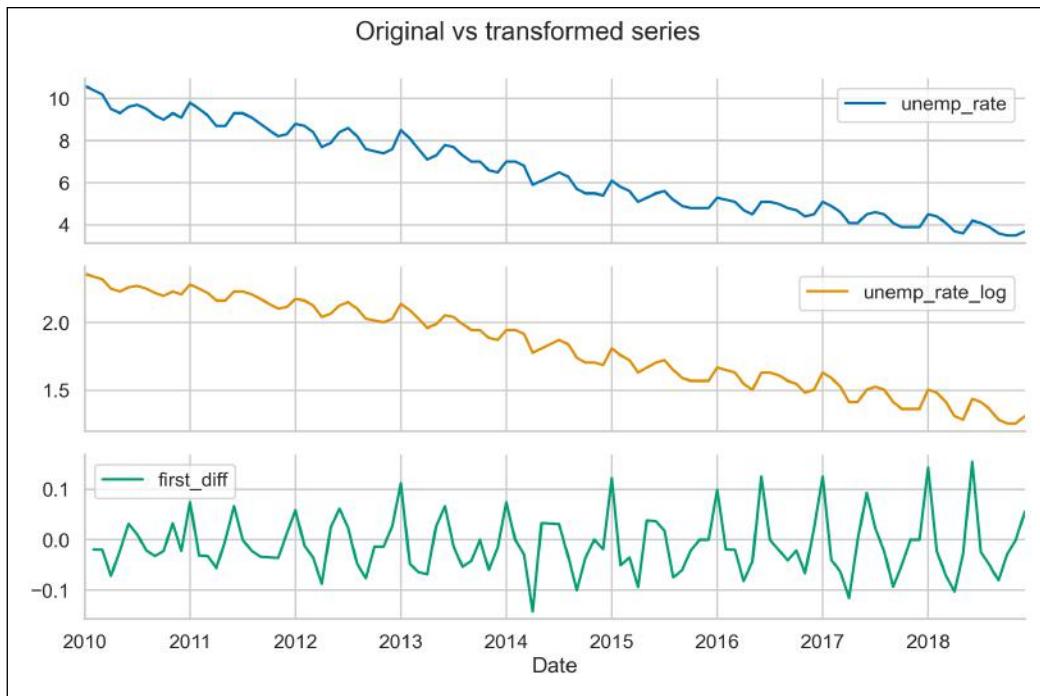


Figure 6.18: Applying transformations to achieve stationarity

4. Test the stationarity of the differenced series:

```
fig = test_acf_pacf(df_train["first_diff"].dropna())
```

Running the function produces the following output:

```
ADF test statistic: -2.97 (p-val: 0.04)
KPSS test statistic: 0.04 (p-val: 0.10)
```

By analyzing the test results, we can state that the first differences of the log transformed series are stationary. We also look at the corresponding autocorrelation plots.

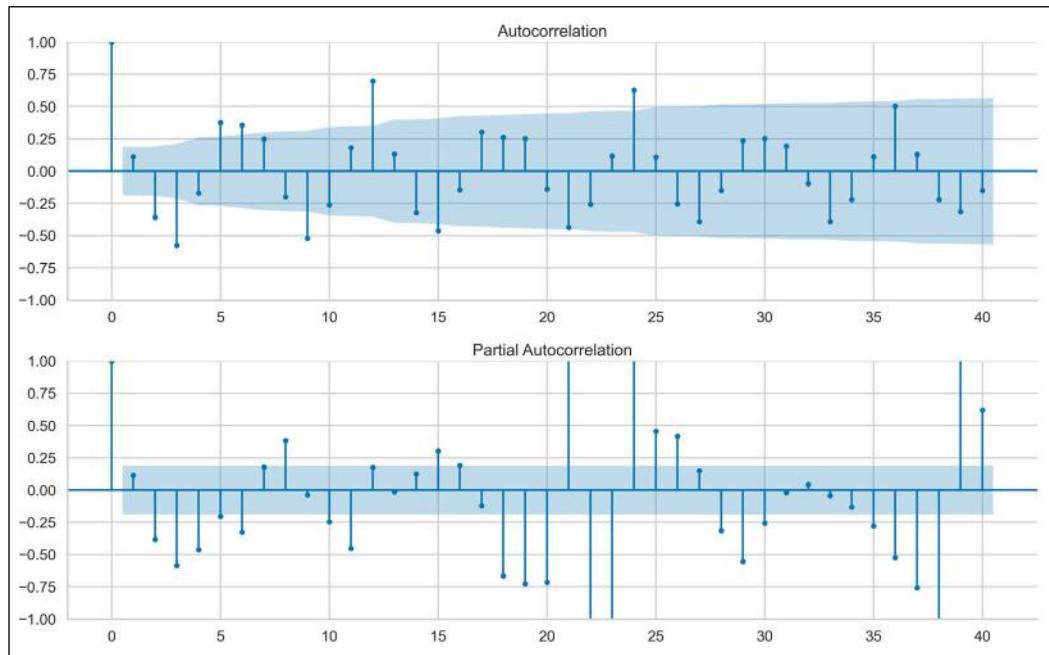


Figure 6.19: The autocorrelation plots of the first differences of the log transformed series

5. Fit two different ARIMA models and print their summaries:

```
arima_111 = ARIMA(  
    df_train["unemp_rate_log"], order=(1, 1, 1)  
).fit()  
arima_111.summary()
```

Running the snippet generates the following summary:

SARIMAX Results							
Dep. Variable:	unemp_rate_log	No. Observations:	108				
Model:	ARIMA(1, 1, 1)	Log Likelihood			157.020		
Date:	Thu, 12 May 2022	AIC			-308.040		
Time:	00:14:40	BIC			-300.021		
Sample:	01-01-2010 - 12-01-2018	HQIC			-304.789		
Covariance Type:	opg						
	coef	std err	z	P> z	[0.025	0.975]	
ar.L1	0.5541	0.401	1.381	0.167	-0.232	1.340	
ma.L1	-0.7314	0.306	-2.391	0.017	-1.331	-0.132	
sigma2	0.0031	0.000	6.823	0.000	0.002	0.004	
Ljung-Box (L1) (Q):	2.55	Jarque-Bera (JB):		9.59			
Prob(Q):	0.11	Prob(JB):		0.01			
Heteroskedasticity (H):	2.33	Skew:		0.64			
Prob(H) (two-sided):	0.01	Kurtosis:		3.71			

Figure 6.20: The summary of the fitted ARIMA(1,1,1) model

The first model was a vanilla ARIMA(1,1,1). For the second one, we go with ARIMA(2,1,2).

```
arima_212 = ARIMA(
    df_train["unemp_rate_log"], order=(2, 1, 2)
).fit()
arima_212.summary()
```

Running the snippet generates the following summary:

SARIMAX Results						
Dep. Variable:	unemp_rate_log		No. Observations:	108		
Model:	ARIMA(2, 1, 2)		Log Likelihood	196.744		
Date:	Thu, 12 May 2022		AIC	-383.488		
Time:	00:14:40		BIC	-370.124		
Sample:	01-01-2010 - 12-01-2018		HQIC	-378.070		
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
ar.L1	0.9952	0.014	69.577	0.000	0.967	1.023
ar.L2	-0.9893	0.014	-69.952	0.000	-1.017	-0.962
ma.L1	-1.1437	38.973	-0.029	0.977	-77.530	75.243
ma.L2	1.0000	68.159	0.015	0.988	-132.589	134.589
sigma2	0.0014	0.093	0.015	0.988	-0.181	0.184
Ljung-Box (L1) (Q):	11.30	Jarque-Bera (JB):	3.25			
Prob(Q):	0.00	Prob(JB):	0.20			
Heteroskedasticity (H):	1.99	Skew:	0.40			
Prob(H) (two-sided):	0.04	Kurtosis:	3.28			

Figure 6.21: The summary of the fitted ARIMA(2,1,2) model

6. Combine the fitted values with the predictions:

```

df["pred_111_log"] = (
    arima_111
    .fittedvalues
    .append(arima_111.forecast(TEST_LENGTH))
)
df["pred_111"] = np.exp(df["pred_111_log"])

df["pred_212_log"] = (
    arima_212
    .fittedvalues
    .append(arima_212.forecast(TEST_LENGTH))
)
df["pred_212"] = np.exp(df["pred_212_log"])
df

```

Running the snippet generates the following table:

	unemp_rate	pred_111_log	pred_111	pred_212_log	pred_212
Date					
2010-01-01	10.6	0.000000	1.000000	0.000000	1.000000
2010-02-01	10.4	2.360854	10.600000	2.360854	10.600000
2010-03-01	10.2	2.344579	10.428881	2.337674	10.357115
2010-04-01	9.5	2.327491	10.252189	2.324158	10.218076
2010-05-01	9.3	2.266964	9.650060	2.256346	9.548141
...
2019-08-01	3.8	1.308394	3.700225	1.340842	3.822261
2019-09-01	3.3	1.308394	3.700226	1.272401	3.569413
2019-10-01	3.3	1.308394	3.700227	1.222022	3.394045
2019-11-01	3.3	1.308394	3.700227	1.239597	3.454222
2019-12-01	3.4	1.308394	3.700227	1.306928	3.694807

Figure 6.22: The predictions from the ARIMA models—raw and transformed back to the original scale

7. Plot the forecasts and calculate the MAPEs:

```
(  
    df[["unemp_rate", "pred_111", "pred_212"]]  
    .iloc[1:]  
    .plot(title="ARIMA forecast of the US unemployment rate")  
)
```

Running the snippet generates the following figure:

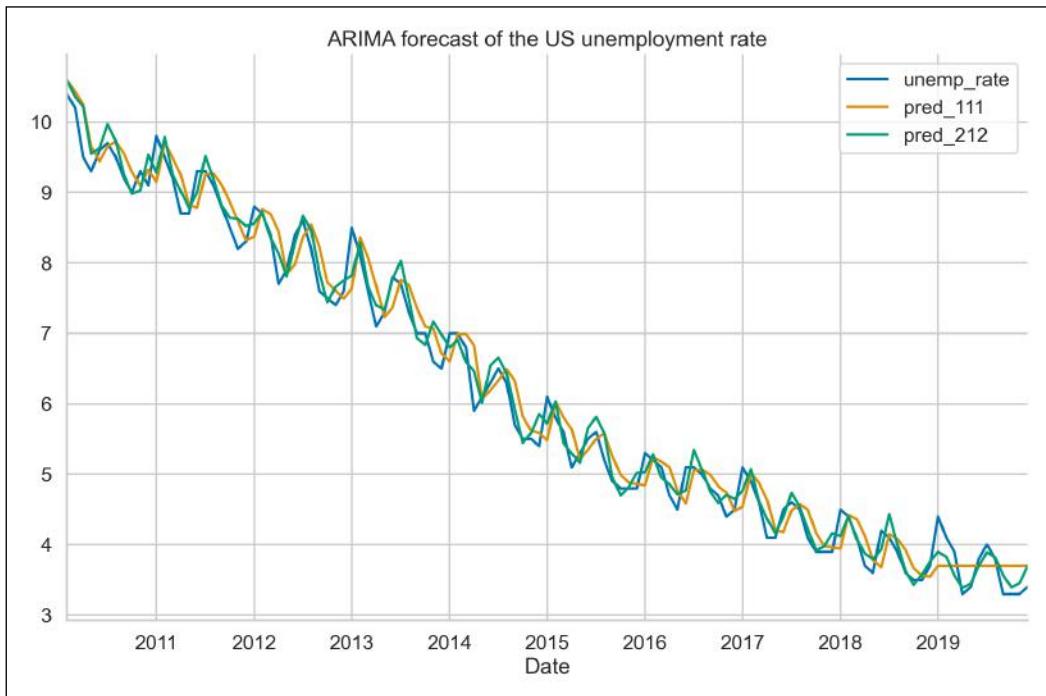


Figure 6.23: The forecast and the fitted values from the two ARIMA models

Now we also zoom into the test set, to clearly see the forecasts:

```
(  
    df[["unemp_rate", "pred_111", "pred_212"]]  
    .iloc[-TEST_LENGTH:]  
    .plot(title="Zooming in on the out-of-sample forecast")  
)
```

Running the snippet generates the following figure:

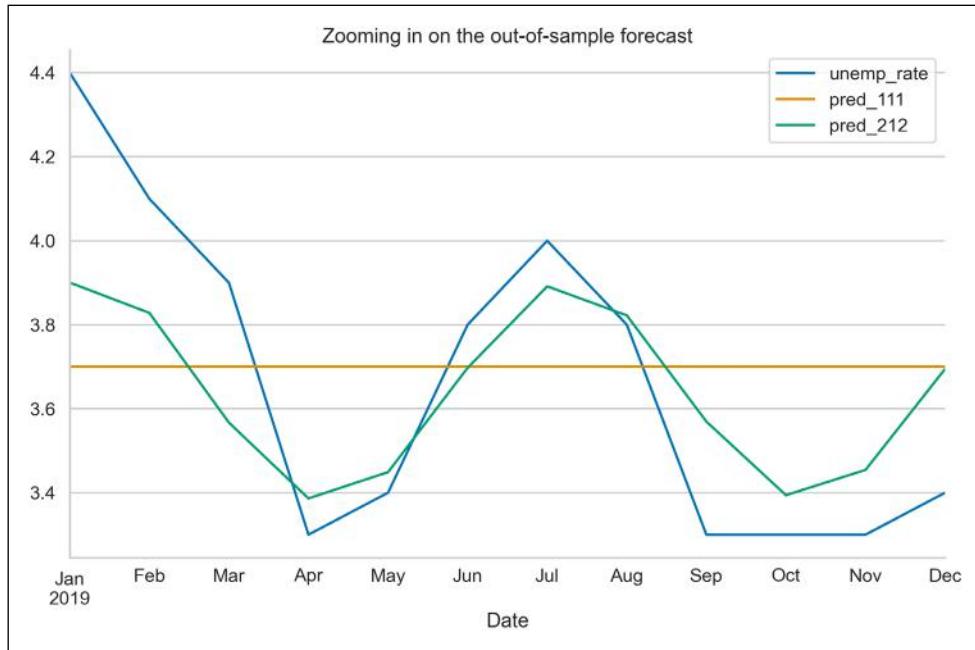


Figure 6.24: The forecast from the two ARIMA models

In Figure 6.24, we can see that the forecast of the ARIMA(1,1,1) is virtually a straight line, while ARIMA(2,1,2) did a better job at capturing the pattern of the original series.

Now we calculate the MAPEs:

```

mape_111 = mean_absolute_percentage_error(
    df["unemp_rate"].iloc[-TEST_LENGTH:],
    df["pred_111"].iloc[-TEST_LENGTH:]
)

mape_212 = mean_absolute_percentage_error(
    df["unemp_rate"].iloc[-TEST_LENGTH:],
    df["pred_212"].iloc[-TEST_LENGTH:]
)

print(f"MAPE of ARIMA(1,1,1): {100 * mape_111:.2f}%")
print(f"MAPE of ARIMA(2,1,2): {100 * mape_212:.2f}%")

```

Running the snippet generates the following output:

```

MAPE of ARIMA(1,1,1): 9.14%
MAPE of ARIMA(2,1,2): 5.08%

```

8. Extract the forecast with the corresponding confidence intervals and plot them all together:

```
preds_df = arima_212.get_forecast(TEST_LENGTH).summary_frame()
preds_df.columns = ["fcst", "fcst_se", "ci_lower", "ci_upper"]
plot_df = df_test[["unemp_rate"]].join(np.exp(preds_df))

fig, ax = plt.subplots()

(
    plot_df[["unemp_rate", "fcst"]]
    .plot(ax=ax,
          title="ARIMA(2,1,2) forecast with confidence intervals")
)

ax.fill_between(plot_df.index,
                plot_df["ci_lower"],
                plot_df["ci_upper"],
                alpha=0.3,
                facecolor="g")

ax.legend(loc="upper left")
```

Running the snippet generates the following figure:

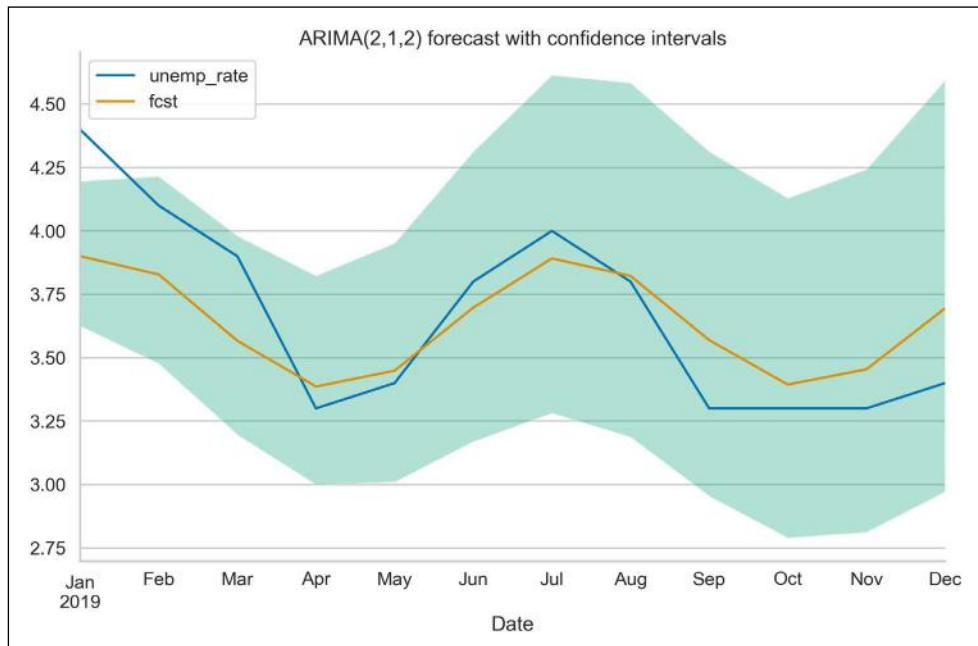


Figure 6.25: The forecast from the ARIMA(2,1,2) model together with its confidence intervals

We can see that the forecast is following the shape of the observed values. Additionally, we can see a typical cone-like pattern of the confidence intervals—the longer the horizon of the forecast, the wider the confidence intervals, which correspond to the increasing uncertainty.

How it works...

After creating the training and test sets in *Step 2*, we applied the log transformation and first differences to the training data.



If we want to apply differencing to a given series more than once, we should use the `np.diff` function as it implements recursive differencing. Using the `diff` method of a DataFrame/Series with `periods > 1` results in taking the difference between the current observations and the one from that many `periods` before.

In *Step 4*, we tested the stationarity of the first differences of the log transformed series. To do so, we used the custom `test_autocorrelation` function. By looking at the outputs of the statistical tests, we see that the series is stationary at the 5% significance level.

When looking at the ACF/PACF plots, we can also clearly see the yearly seasonal pattern (at lags 12 and 24).

In *Step 5*, we fitted two ARIMA models: ARIMA(1,1,1) and ARIMA(2,1,2). First, the series turned out to be stationary after the first differences, so we knew that the order of integration was $d=1$. Normally, we can use the following set of “rules” to determine the values of p and q .

Identifying the order of the AR model:

- The ACF shows a significant autocorrelation up to lag p and then trails off afterward
- As the PACF only describes the direct relationship between an observation and its lag, we would expect no significant correlations beyond lag p

Identifying the order of the MA model:

- The PACF shows a significant autocorrelation up to lag q and then trails off afterward
- The ACF shows significant autocorrelation coefficients up to lag q and then will exhibit a sharp decline

Regarding the manual calibration of ARIMA’s orders, Hyndman and Athanasopoulos (2018) warned that if both p and q are positive, the ACF/PACF plots might not be helpful in determining the specification of the ARIMA model. In the next recipe, we will introduce an automatic approach to determine the optimal values of ARIMA hyperparameters.

In *Step 6*, we combined the original series with the predictions from the two models. We extracted the fitted values from the ARIMA models and appended the forecasts for 2019 to the end of the series. Because we fitted the models to the log transformed series, we had to reverse the transformation by using the exponent function (`np.exp`).



When working with series that can have 0 values, it is safer to use `np.log1p` and `np.exp1m`. This way, we avoid potential errors when taking the logarithm of 0.

In Step 7, we plotted the forecasts and calculated the mean absolute percentage error. The ARIMA(2,1,2) provided much better forecasts than the simple ARIMA(1,1,1).

In Step 8, we chained the `get_forecast` method of the fitted ARIMA model together with the `summary_frame` method to obtain the forecast and its corresponding confidence intervals. We had to use the `get_forecast` method, as the `forecast` method only returns point forecasts, without any additional information. Lastly, we renamed the columns and plotted them together with the original series.

There's more...

We have already fitted the ARIMA models and explored the accuracy of their forecasts. However, we can also investigate some goodness-of-fit criteria of the fitted models. Instead of focusing on the out-of-sample performance, we can dive a bit deeper into how well the models fit the training data. We do so by looking at the residuals of the fitted ARIMA models.

First, we plot diagnostic plots for the residuals of the fitted ARIMA(2,1,2) model:

```
arima_212.plot_diagnostics(figsize=(18, 14), lags=25)
```

Running the snippet generates the following figure:

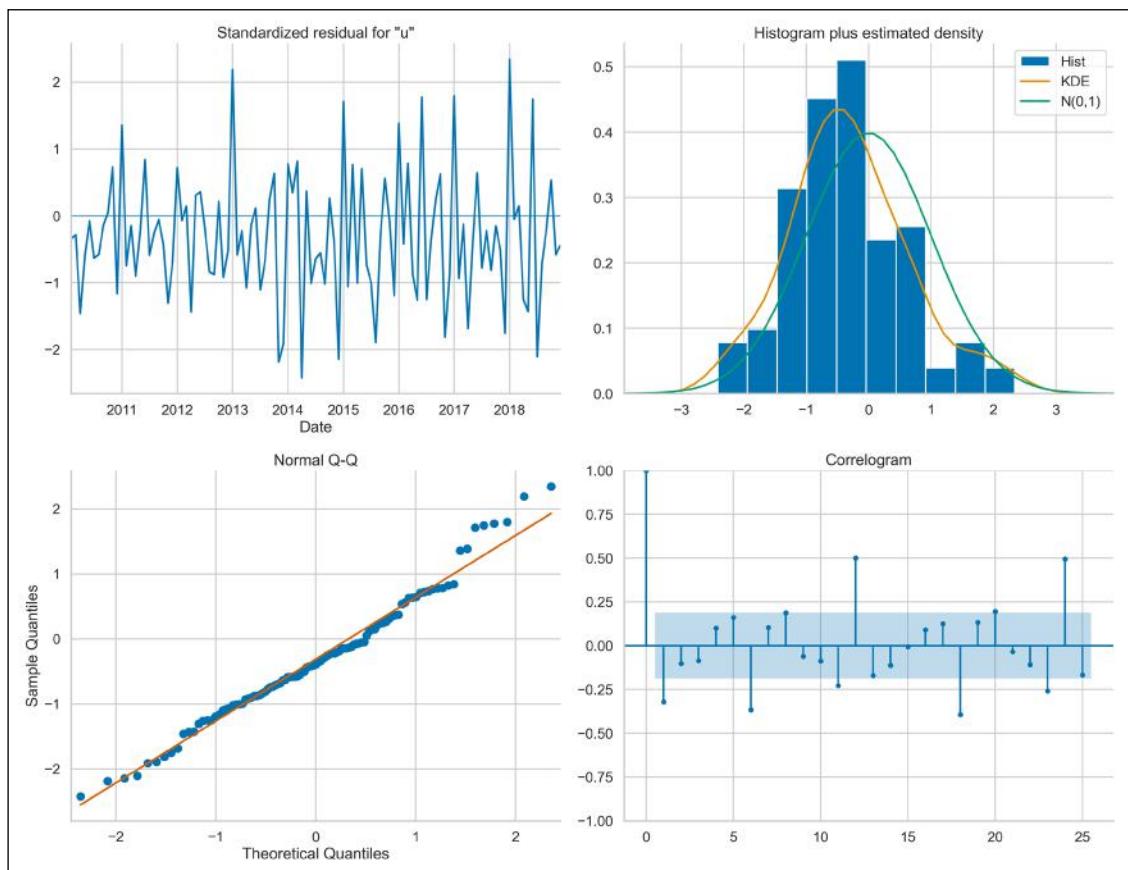


Figure 6.26: The diagnostics plot of the fitted ARIMA(2,1,2) model

Below we cover the interpretation of each of the plots:

- Standardized residuals over time (top left)—the residuals should behave like white noise, that is, there should be no clear patterns visible. Also, the residuals should have a mean of zero and a uniform variance. In our case, there seem to be more negative values than positive ones, so the mean is also probably negative.
- The histogram and the KDE estimate (top right)—the KDE curve of the residuals should be very similar to the one of the standard normal distribution (labeled as $N(0,1)$). We can see that this is not the case for our model, as the distribution is shifted toward the negative values.
- Q-Q plot (bottom left)—the majority of the data points should lie in a straight line. This would indicate that the quantiles of a theoretical distribution (Standard Normal) match the empirical ones. Significant deviations from the diagonal line imply that the empirical distribution is skewed.
- Correlogram (bottom right)—here we are looking at the plot of the autocorrelation function of the residuals. We would expect that the residuals of a well-fitted ARIMA model are not auto-correlated. In our case, we can clearly see correlated residuals at lags 12 and 24. This is a hint that the model is not capturing the seasonal patterns present in the data.

To continue investigating the autocorrelation of the residuals, we can also apply Ljung-Box's test for no autocorrelation. To do so, we use the `test_serial_correlation` method of the fitted ARIMA model. Alternatively, we could use the `acorr_ljungbox` function from `statsmodels`.

```
ljung_box_results = arima_212.test_serial_correlation(method="ljungbox")
ljung_box_pvals = ljung_box_results[0][1]

fig, ax = plt.subplots(1, figsize=[16, 5])
sns.scatterplot(x=range(len(ljung_box_pvals)),
                 y=ljung_box_pvals,
                 ax=ax)
ax.axhline(0.05, ls="--", c="r")
ax.set(title="Ljung-Box test's results",
       xlabel="Lag",
       ylabel="p-value")
```

Running the snippet generates the following figure:

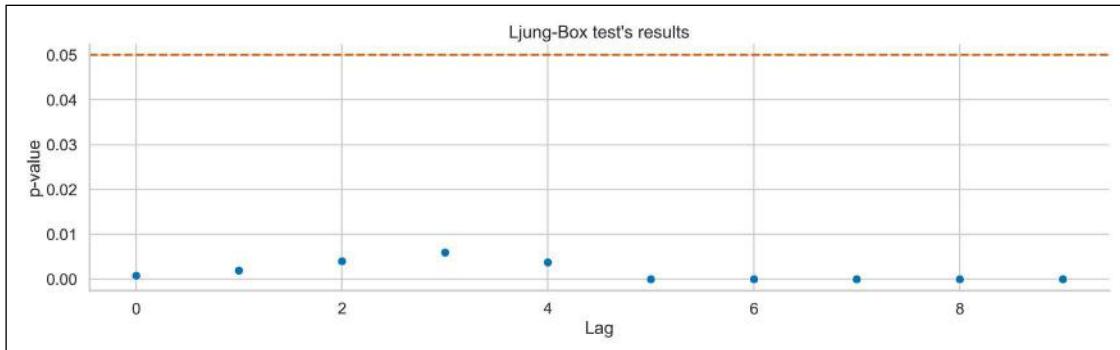


Figure 6.27: The results of the Ljung-Box test for no autocorrelation in the residuals

All of the returned p-values are below the 5% significance level, which means we should reject the null hypothesis stating there is no autocorrelation in the residuals. It makes sense, as we have already observed a significant yearly correlation caused by the fact that our model is missing the seasonal patterns.

What we should also keep in mind is the number of lags to investigate while performing the Ljung-Box test. Different sources suggest a different number of lags to consider. The default value in `statsmodels` is `min(10, nobs // 5)` for non-seasonal models and `min(2*m, nobs // 5)` for seasonal time series, where `m` denotes the seasonal period. Other commonly used variants include `min(20, nobs - 1)` and `ln(nobs)`. In our case, we did not use a seasonal model, so the default value is 10. But as we know, the data does exhibit seasonal patterns, so we should have looked at more lags.

The fitted ARIMA models also contain the `test_normality` and `test_heteroskedasticity` methods, which we could use for further evaluation of the model's fit. We leave exploring those as an exercise for the reader.

See also

Please see the following references for more information on fitting ARIMA models and helpful sets of rules for manually picking up the correct orders of the models:

- <https://online.stat.psu.edu/stat510/lesson/3/3.1>
- <https://people.duke.edu/~rnau/arimrule.htm>

For more information on the Ljung-Box test:

- <https://robjhyndman.com/hyndtsight/ljung-box-test/>

Finding the best-fitting ARIMA model with auto-ARIMA

As we have seen in the previous recipe, the performance of an ARIMA model varies greatly depending on the chosen hyperparameters (p , d , and q). We can do our best to choose them based on our intuition, the statistical tests, and the ACF/PACF plots. However, this can prove to be quite difficult to do in practice.

That is why in this recipe we introduce **auto-ARIMA**, an automated approach to finding the best hyperparameters of the ARIMA class models (including variants such as ARIMAX and SARIMA).

Without going much into the technical details of the algorithm, it first determines the number of differences using the KPSS test. Then, the algorithm uses a stepwise search to traverse the model space, searching for a model that results in a better fit. A popular choice of evaluation metric used for comparing the models is the **Akaike Information Criterion (AIC)**. The metric provides a trade-off between the goodness of fit of the model and its simplicity—AIC deals with the risks of overfitting and underfitting. When we compare multiple models, the lower the value of AIC, the better the model. For a more complete description of the auto-ARIMA procedure, please refer to the sources mentioned in the *See also* section.

The auto-ARIMA framework also works well with the extensions of the ARIMA model:

- **ARIMAX**—adds exogenous variable(s) to the model.
- **SARIMA (Seasonal ARIMA)**—extends ARIMA to account for seasonality in the time series. The full specification is $\text{SARIMA}(p,d,q)(P,D,Q)m$, where the capitalized parameters are analogous to the original ones, but they refer to the seasonal component of the time series. m refers to the period of seasonality.

In this recipe, we will once again work with the monthly US unemployment rates from the years 2010 to 2019.

Getting ready

We will use the same data that we used in the *Time series decomposition* recipe.

How to do it...

Execute the following steps to find the best-fitting ARIMA model using the auto-ARIMA procedure:

1. Import the libraries:

```
import pandas as pd
import pmdarima as pm
from sklearn.metrics import mean_absolute_percentage_error
```

2. Create the train/test split:

```
TEST_LENGTH = 12
df_train = df.iloc[:-TEST_LENGTH]
df_test = df.iloc[-TEST_LENGTH:]
```

3. Find the best hyperparameters of the ARIMA model using the auto-ARIMA procedure:

```
auto_arima = pm.auto_arima(df_train,
                           test="adf",
                           seasonal=False,
                           with_intercept=False,
                           stepwise=True,
                           suppress_warnings=True,
                           trace=True)

auto_arima.summary()
```

Executing the snippet generates the following summary:

SARIMAX Results						
Dep. Variable:	y	No. Observations:	108			
Model:	SARIMAX(2, 1, 2)	Log Likelihood	1.294			
Date:	Thu, 12 May 2022	AIC	7.411			
Time:	00:14:44	BIC	20.775			
Sample:	0	HQIC	12.829			
	- 108					
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
ar.L1	0.9882	0.028	34.788	0.000	0.933	1.044
ar.L2	-0.9630	0.023	-41.667	0.000	-1.008	-0.918
ma.L1	-1.1926	0.057	-21.093	0.000	-1.303	-1.082
ma.L2	0.9241	0.063	14.580	0.000	0.800	1.048
sigma2	0.0550	0.009	6.065	0.000	0.037	0.073
Ljung-Box (L1) (Q):	5.22	Jarque-Bera (JB):	4.47			
Prob(Q):	0.02	Prob(JB):	0.11			
Heteroskedasticity (H):	0.52	Skew:	0.44			
Prob(H) (two-sided):	0.05	Kurtosis:	3.48			

Figure 6.28: The summary of the best-fitting ARIMA model, as identified using the auto-ARIMA procedure

The procedure indicated that the best-fitting ARIMA model is ARIMA(2,1,2). But as you can see, the results in *Figure 6.28* and *Figure 6.21* are different. That is because in the latter case, we have fitted the ARIMA(2,1,2) model to the log transformed series, while in this recipe, we have not applied the log transformation.

Because we indicated `trace=True`, we also see the following information about the models fitted during the procedure:

```
Performing stepwise search to minimize aic
ARIMA(2,1,2)(0,0,0)[0] : AIC=7.411, Time=0.24 sec
ARIMA(0,1,0)(0,0,0)[0] : AIC=77.864, Time=0.01 sec
ARIMA(1,1,0)(0,0,0)[0] : AIC=77.461, Time=0.01 sec
ARIMA(0,1,1)(0,0,0)[0] : AIC=75.688, Time=0.01 sec
ARIMA(1,1,2)(0,0,0)[0] : AIC=68.551, Time=0.01 sec
ARIMA(2,1,1)(0,0,0)[0] : AIC=54.321, Time=0.03 sec
ARIMA(3,1,2)(0,0,0)[0] : AIC=7.458, Time=0.07 sec
ARIMA(2,1,3)(0,0,0)[0] : AIC=inf, Time=0.07 sec
ARIMA(1,1,1)(0,0,0)[0] : AIC=78.507, Time=0.02 sec
ARIMA(1,1,3)(0,0,0)[0] : AIC=60.069, Time=0.02 sec
ARIMA(3,1,1)(0,0,0)[0] : AIC=41.703, Time=0.02 sec
ARIMA(3,1,3)(0,0,0)[0] : AIC=10.527, Time=0.10 sec
ARIMA(2,1,2)(0,0,0)[0] intercept : AIC=inf, Time=0.08 sec

Best model: ARIMA(2,1,2)(0,0,0)[0]
Total fit time: 0.740 seconds
```

Similar to the ARIMA model estimated with the `statsmodels` library, with `pmdarima` (which is in fact a wrapper around `statsmodels`) we can also use the `plot_diagnostics` method to analyze the fit of the model by looking at its residuals:

```
auto_arima.plot_diagnostics(figsize=(18, 14), lags=25)
```

Executing the snippet generates the following figure:

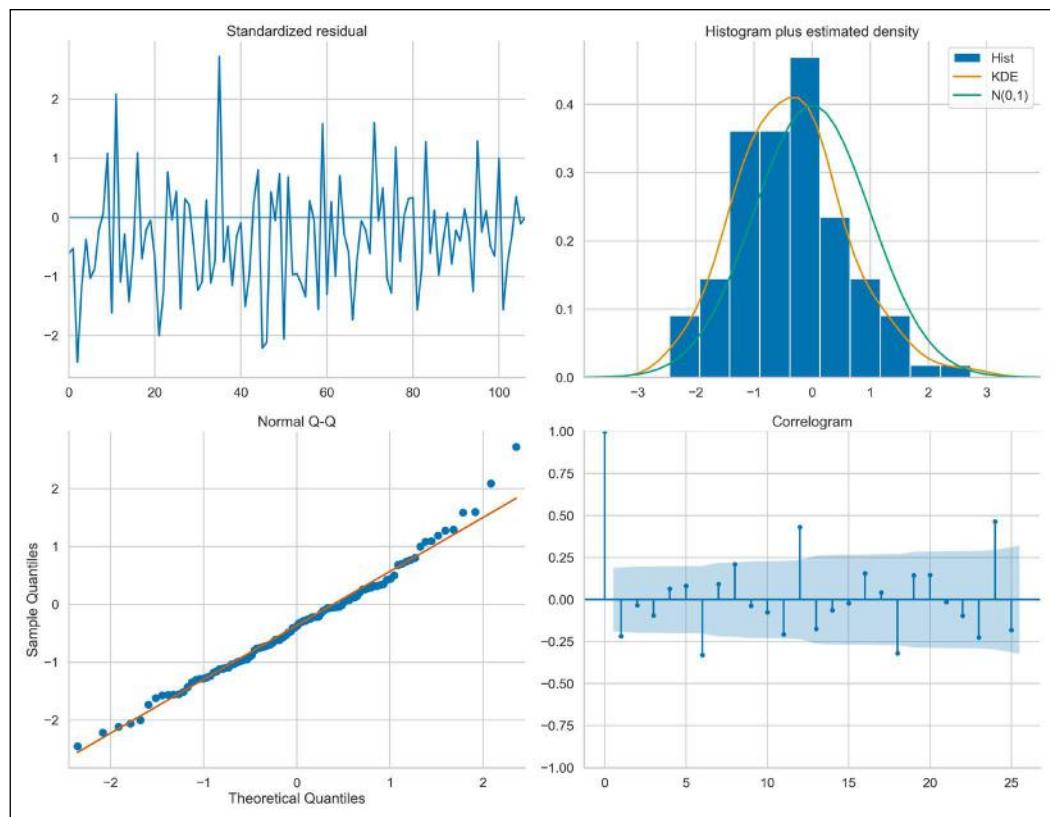


Figure 6.29: The diagnostic plots of the best-fitting ARIMA model

Similar to the diagnostics plot in *Figure 6.26*, this ARIMA(2,1,2) model is also struggling with capturing the yearly seasonal patterns—we can clearly see that in the correlogram.

4. Find the best hyperparameters of a SARIMA model using the auto-ARIMA procedure:

Executing the snippet generates the following summary:

SARIMAX Results						
Dep. Variable:	y	No. Observations:	108			
Model:	SARIMAX(0, 1, 5)x(2, 0, [1], 12)	Log Likelihood	41.060			
Date:	Thu, 12 May 2022	AIC	-64.120			
Time:	00:15:12	BIC	-40.065			
Sample:	0	HQIC	-54.368			
	- 108					
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
ma.L1	-0.2808	0.115	-2.449	0.014	-0.506	-0.056
ma.L2	-0.1211	0.112	-1.078	0.281	-0.341	0.099
ma.L3	-0.2031	0.129	-1.575	0.115	-0.456	0.050
ma.L4	0.0749	0.122	0.616	0.538	-0.164	0.313
ma.L5	0.2473	0.105	2.363	0.018	0.042	0.452
ar.S.L12	0.8702	0.160	5.455	0.000	0.558	1.183
ar.S.L24	0.1275	0.152	0.838	0.402	-0.171	0.426
ma.S.L12	-0.8719	0.344	-2.536	0.011	-1.546	-0.198
sigma2	0.0200	0.006	3.589	0.000	0.009	0.031
Ljung-Box (L1) (Q):	0.18	Jarque-Bera (JB):	1.98			
Prob(Q):	0.67	Prob(JB):	0.37			
Heteroskedasticity (H):	0.60	Skew:	-0.29			
Prob(H) (two-sided):	0.13	Kurtosis:	3.31			

Figure 6.30: The summary of the best-fitting SARIMA model, as identified using the auto-ARI-MA procedure

Just as we have done before, we will also look at the various residual plots:

```
auto_sarima.plot_diagnostics(figsize=(18, 14), lags=25)
```

Executing the snippet generates the following figure:

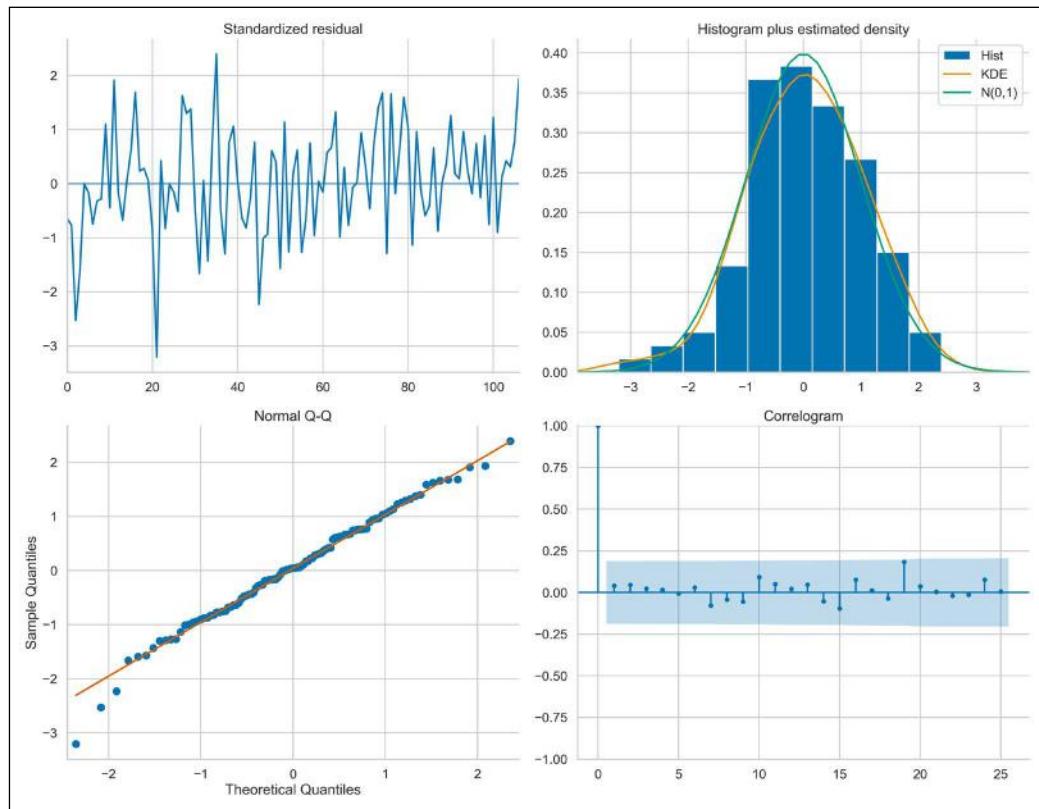


Figure 6.31: The diagnostic plots of the best-fitting SARIMA model

We can clearly see that the SARIMA model results in a much better fit than the ARIMA(2,1,2) model.

5. Calculate the forecasts from the two models and plot them:

```
df_test["auto_arima"] = auto_arima.predict(TEST_LENGTH)
df_test["auto_sarima"] = auto_sarima.predict(TEST_LENGTH)
df_test.plot(title="Forecasts of the best ARIMA/SARIMA models")
```

Executing the snippet generates the following plot:

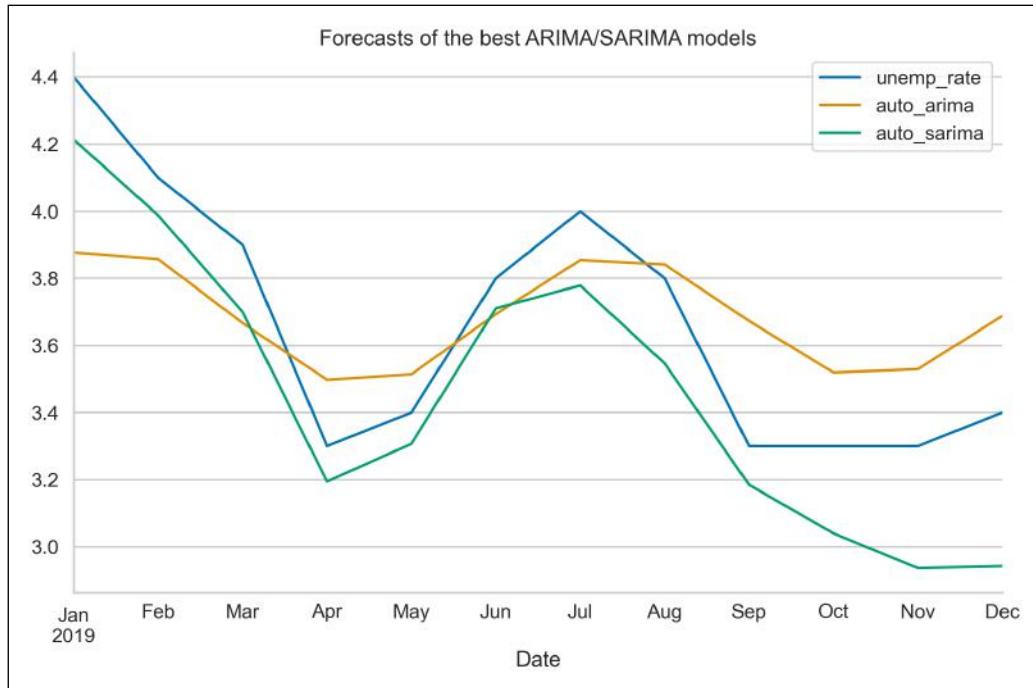


Figure 6.32: The forecasts from the ARIMA and SARIMA models identified using the auto-ARIMA procedure

It should not come as a surprise that the SARIMA model is capturing the seasonal patterns better than the ARIMA model. That is also reflected in the performance metrics calculated below. We also calculate the MAPEs:

```

mape_auto_arima = mean_absolute_percentage_error(
    df_test["unemp_rate"],
    df_test["auto_arima"]
)

mape_auto_sarima = mean_absolute_percentage_error(
    df_test["unemp_rate"],
    df_test["auto_sarima"]
)

print(f"MAPE of auto-ARIMA: {100*mape_auto_arima:.2f}%")
print(f"MAPE of auto-SARIMA: {100*mape_auto_sarima:.2f}%")

```

Executing the snippet generates the following output:

```
MAPE of auto-ARIMA: 6.17%
MAPE of auto-SARIMA: 5.70%
```

How it works...

After importing the libraries, we created the training and test set just as we have done in the previous recipes.

In *Step 3*, we used the `auto_arima` function to find the best hyperparameters of the ARIMA model. While using it, we specified that:

- We wanted to use the augmented Dickey-Fuller test as the stationarity test instead of the KPSS test.
- We turned off the seasonality to fit an ARIMA model instead of SARIMA.
- We wanted to estimate a model without an intercept, which is also the default setting when estimating ARIMA in `statsmodels` (under the `trend` argument in the `ARIMA` class).
- We wanted to use the stepwise algorithm for identifying the best hyperparameters. When we mark this one as `False`, the function will run an exhaustive grid search (trying out all possible hyperparameter combinations) similarly to the `GridSearchCV` class of `scikit-learn`. When using that scenario, we can indicate the `n_jobs` argument to specify how many models can be fitted in parallel.

There are also many different settings we can experiment with, for example:

- Selecting the starting value of the hyperparameters for the search.
- Capping the maximum values of parameters in the search.
- Selecting different statistical tests for determining the number of differences (also seasonal).
- Selecting an out-of-sample evaluation period (`out_of_sample_size`). This will make the algorithm fit the models on the data up until a certain point in time (the last observation minus `out_of_sample_size`) and evaluate on the hold-out set. This way of selecting the best model might be preferable when we care more about the forecasting performance than the fit to the training data.
- We can cap the maximum time for fitting the model or the max number of hyperparameter combinations to try out. This is especially useful when estimating seasonal models on more granular (for example, weekly) data, as such scenarios tend to take quite a long time to fit.

In *Step 4*, we used the `auto_arima` function to find the best SARIMA model. To do so, we specified `seasonal=True` and indicated that we are working with monthly data by setting `m=12`.

Lastly, we calculated the forecasts coming from the two models using the `predict` method, plotted them together with the ground truth, and calculated the MAPEs.

There's more...

We can use the auto-ARIMA framework in the `pmdarima` library to estimate even more complex models or entire pipelines, which include transforming the target variable or adding new features. In this section, we show how to do so.

We start by importing a few more classes:

```
from pmdarima.pipeline import Pipeline
from pmdarima.preprocessing import FourierFeaturizer
from pmdarima.preprocessing import LogEndogTransformer
from pmdarima import arima
```

For the first model, we train an ARIMA model with additional features (exogenous variables). As an experiment, we try to provide features indicating which month a given observation comes from. If this works, we might not need to estimate a SARIMA model to capture the yearly seasonality.

We create **dummy variables** using the `pd.get_dummies` function. Each column contains a Boolean flag indicating if the observation came from the given month or not.

We also need to drop the first column from the new DataFrame to avoid the **dummy-variable trap** (perfect multicollinearity). We added the new variables for both the training and test sets:

```
month_dummies = pd.get_dummies(
    df.index.month,
    prefix="month_",
    drop_first=True
)
month_dummies.index = df.index
df = df.join(month_dummies)

df_train = df.iloc[:-TEST_LENGTH]
df_test = df.iloc[-TEST_LENGTH:]
```

We then use the `auto_arimax` function to find the best-fitting model. The only thing that changes as compared to *Step 3* of this recipe is that we had to specify the exogenous variables using the `exogenous` argument. We indicated all columns except the one containing the target. Alternatively, we could have kept the additional variables in a separate object with identical indices as the target:

```
auto_arimax = pm.auto_arima(
    df_train[["unemp_rate"]],
    exogenous=df_train.drop(columns=["unemp_rate"]),
    test="adf",
    seasonal=False,
    with_intercept=False,
    stepwise=True,
```

```

    suppress_warnings=True,
    trace=True
)

auto_arimax.summary()

```

Executing the snippet generates the following summary:

SARIMAX Results						
Dep. Variable:	y	No. Observations:	108			
Model:	SARIMAX(0, 1, 2)	Log Likelihood	64.959			
Date:	Sat, 30 Jul 2022	AIC	-99.917			
Time:	00:02:14	BIC	-59.825			
Sample:	01-01-2010 - 12-01-2018	HQIC	-83.664			
Covariance Type:	opg					
	coef	std err	z	P> z	[0.025	0.975]
intercept	-0.0590	0.009	-6.760	0.000	-0.076	-0.042
month_2	-0.1329	0.072	-1.856	0.063	-0.273	0.007
month_3	-0.3405	0.074	-4.593	0.000	-0.486	-0.195
month_4	-0.8482	0.066	-12.846	0.000	-0.978	-0.719
month_5	-0.7560	0.067	-11.337	0.000	-0.887	-0.625
month_6	-0.2637	0.066	-3.996	0.000	-0.393	-0.134
month_7	-0.1491	0.074	-2.016	0.044	-0.294	-0.004
month_8	-0.3345	0.070	-4.778	0.000	-0.472	-0.197
month_9	-0.6422	0.068	-9.434	0.000	-0.776	-0.509
month_10	-0.7278	0.067	-10.938	0.000	-0.858	-0.597
month_11	-0.7578	0.055	-13.712	0.000	-0.866	-0.649
month_12	-0.6761	0.045	-15.036	0.000	-0.764	-0.588
ma.L1	-0.2874	0.106	-2.711	0.007	-0.495	-0.080
ma.L2	-0.1234	0.105	-1.179	0.239	-0.329	0.082
sigma2	0.0174	0.003	6.427	0.000	0.012	0.023
Ljung-Box (L1) (Q):	0.01	Jarque-Bera (JB):	0.05			
Prob(Q):	0.91	Prob(JB):	0.98			
Heteroskedasticity (H):	0.60	Skew:	-0.03			
Prob(H) (two-sided):	0.13	Kurtosis:	2.91			

Figure 6.33: The summary of the ARIMA model with exogenous variables

We also look at the residuals plots by using the `plot_diagnostics` method. It seems that the auto-correlation issues connected to the yearly seasonality were fixed by including the dummy variables.

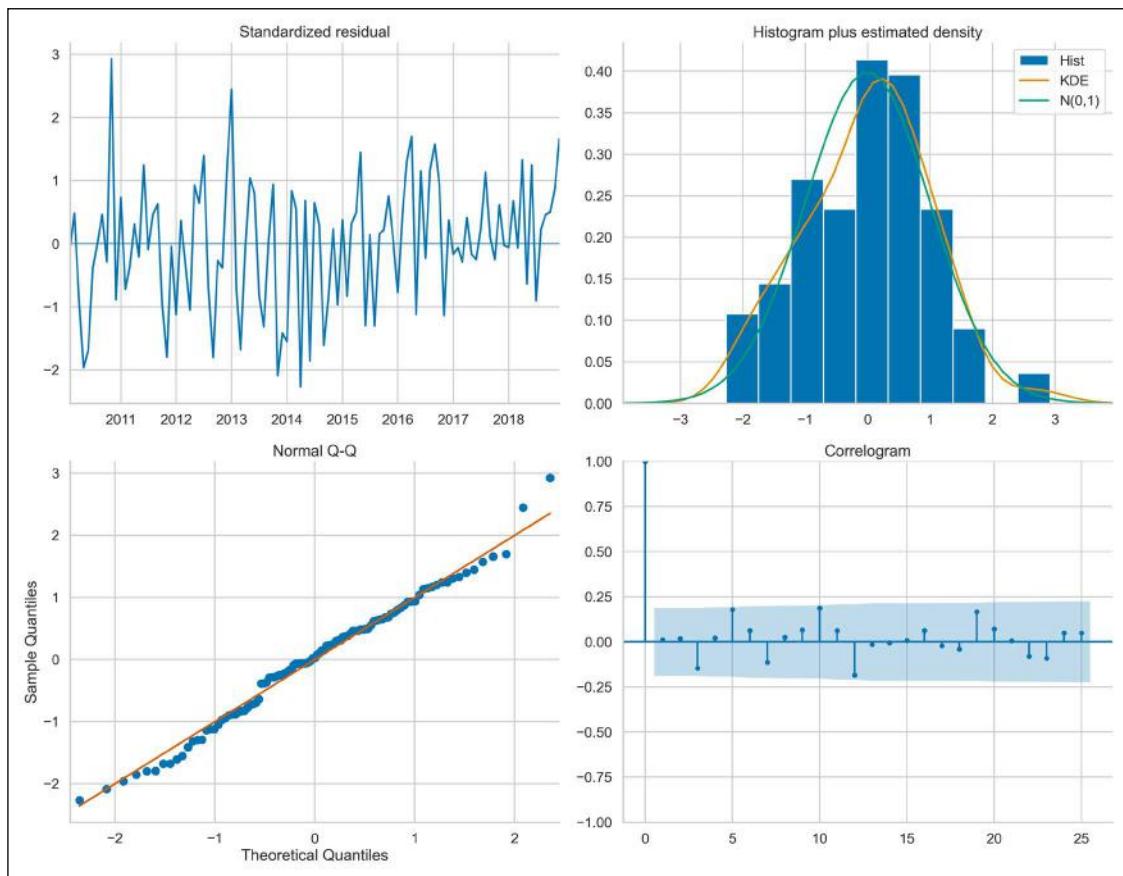


Figure 6.34: The diagnostic plots of the ARIMA model with exogenous variables

Lastly, we also show how to create an entire data transformation and modeling pipeline, which also finds the best-fitting ARIMA model. Our pipeline consists of three steps:

- We apply the log transformation to the target.
- We create new features using the `FourierFeaturizer`—explaining Fourier series is outside of the scope of this book. In practice, using them permits us to account for the seasonality in a seasonal time series without using a seasonal model *per se*. To provide a bit more context, it is something similar to what we have done with the month dummies. The `FourierFeaturizer` class supplies decomposed seasonal Fourier terms as an exogenous array of features. We had to specify the seasonal periodicity m .

- We find the best-fitting model using the auto-ARIMA procedure. Please keep in mind that when using the pipelines, we have to use the AutoARIMA class instead of the `pm.auto_arima` function. Those two offer the same functionalities, just this time we had to use a class to make it compatible with the Pipeline functionality.

```
auto_arima_pipe = Pipeline([
    ("log_transform", LogEndogTransformer()),
    ("fourier", FourierFeaturizer(m=12)),
    ("arima", arima.AutoARIMA(stepwise=True, trace=1,
                               error_action="warn",
                               test="adf", seasonal=False,
                               with_intercept=False,
                               suppress_warnings=True))
])

auto_arima_pipe.fit(df_train[["unemp_rate"]])
```

In the log produced by fitting the pipeline, we can see that the following model was selected as the best one:

```
Best model: ARIMA(4,1,0)(0,0,0)[0] intercept
```

The biggest advantage of using the pipeline is that we do not have to carry out all the steps ourselves. We just define a pipeline and then provide a time series as input to the `fit` method. In general, pipelines (also the ones in `scikit-learn` as we will see in *Chapter 13, Applied Machine Learning: Identifying Credit Default*) are a great functionality that helps us with:

- Making the code reusable
- Defining a clear order of the operations that are happening on the data
- Avoiding potential data leakage when creating features and splitting the data

A potential disadvantage of using the pipelines is that some operations are not that easy to track anymore (the intermediate results are not stored as separate objects) and it is a bit more difficult to access the particular elements of the pipeline. For example, we cannot run `auto_arima_pipe.summary()` to get the summary of the fitted ARIMA model.

Below, we create forecasts using the `predict` method. Some noteworthy things about this step:

- We created a new DataFrame containing only the target. We did so to remove the extra columns we created earlier in this recipe.
- When using the `predict` method with a fitted ARIMAX model, we also need to provide the required exogenous variables for the predictions. They are passed as the `X` argument.

- When we use the predict method of a pipeline that transforms the target variable, the returned predictions (or fitted values) are expressed on the same scale as the original input. In our case, the following sequence happened under the hood. First, the original time series was log transformed. Then, new features were added. Next, we obtained predictions from the model (still on the log transformed scale). Finally, the predictions were converted to the original scale using the exponent function.

```
results_df = df_test[["unemp_rate"]].copy()
results_df["auto_arimax"] = auto_arimax.predict(
    TEST_LENGTH,
    X=df_test.drop(columns=["unemp_rate"])
)
results_df["auto_arima_pipe"] = auto_arima_pipe.predict(TEST_LENGTH)
results_df.plot(title="Forecasts of the ARIMAX/pipe models")
```

Running the code generates the following plot:

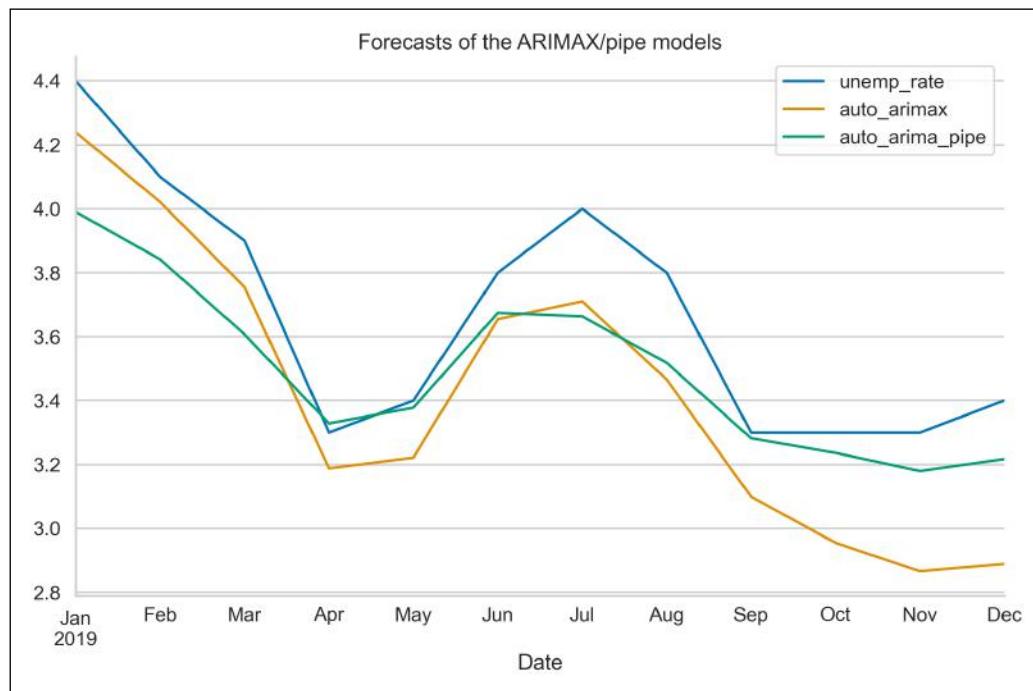


Figure 6.35: The forecasts from the ARIMAX model and the ARIMA pipeline

For reference, we also add the scores of those forecasts:

```
MAPE of auto-ARIMAX: 6.88%
MAPE of auto-pipe: 4.61%
```

Of all the ARIMA models we have tried in this chapter, the pipeline model performed best. However, it still performs significantly worse than the exponential smoothing methods.



When using the `predict` method of the ARIMA models/pipelines in the `pmdarima` library, we can set the `return_conf_int` argument to `True`. When we do so, the method will not only return the point forecast but also the corresponding confidence intervals.

See also

- Hyndman, R. J. & Athanasopoulos, G. 2021. “ARIMA Modeling in Fable.” In *Forecasting: Principles and Practice*, 3rd edition, OTexts: Melbourne, Australia. OTexts.com/fpp3. Accessed on 2022-05-08 – <https://otexts.com/fpp3/arima-r.html>.
- Hyndman, R. J. & Khandakar, Y., 2008. “Automatic time series forecasting: the forecast package for R,” *Journal of Statistical Software*, 27: 1-22.

Summary

In this chapter, we have covered the classical (statistical) approaches to time series analysis and forecasting. We learned how to decompose any time series into trend, seasonal, and remainder components. This step can be very helpful in getting a better understanding of the explored time series. But we can also use it directly for modeling purposes.

Then, we explained how to test if a time series is stationary, as some of the statistical models (for example, ARIMA) require stationarity. We also explained which steps we can take to transform a non-stationary time series into a stationary one.

Lastly, we explored two of the most popular statistical approaches to time series forecasting—exponential smoothing methods and ARIMA models. We have also touched upon more modern approaches to estimating such models, which involve automatic tuning and hyperparameter selection.

In the next chapter, we will explore ML-based approaches to time series forecasting.

7

Machine Learning-Based Approaches to Time Series Forecasting

In the previous chapter, we provided a brief introduction to time series analysis and demonstrated how to use statistical approaches (ARIMA and ETS) for time series forecasting. While those approaches are still very popular, they are somewhat dated. In this chapter, we focus on the more recent, ML-based approaches to time series forecasting.

We start by explaining different ways of validating time series models. Then, we move on to the inputs of ML models, that is, the features. We provide an overview of selected feature engineering approaches and introduce a tool for automatic feature extraction that generates hundreds or thousands of features for us.

Having covered those two topics, we introduce the concept of reduced regression, which allows us to reframe the time series forecasting problem as a regular regression problem. Thus, it allows us to use popular and battle-tested regression algorithms (all the ones available in `scikit-learn`, XGBoost, LightGBM, and so on) for time series forecasting. Then, we also show how to use Meta's Prophet algorithm. We conclude the chapter by introducing one of the popular AutoML tools, which allows us to train and tune dozens of ML models with only a few lines of code.

We cover the following recipes in this chapter:

- Validation methods for time series
- Feature engineering for time series
- Time series forecasting as reduced regression
- Forecasting with Meta's Prophet
- AutoML for time series forecasting with PyCaret

Validation methods for time series

In the previous chapter, we trained a few statistical models to forecast the future values of time series. To evaluate the models' performance, we initially split the data into training and test sets. However, that is definitely not the only approach to model validation.

A very popular approach to evaluating models' performance is called **cross-validation**. It is especially useful for choosing the best set of a model's hyperparameters or selecting the best model for the problem we are trying to solve. Cross-validation is a technique that allows us to obtain reliable estimates of the model's generalization error by providing multiple estimates of the model's performance. As such, cross-validation can help us greatly when we are dealing with smaller datasets.

The basic cross-validation scheme is called **k-fold cross-validation**, in which we randomly split the training data into k folds. Then, we train the model using $k-1$ folds and evaluate the performance on the k th fold. We repeat this process k times and average the resulting scores. *Figure 7.1* illustrates the procedure.

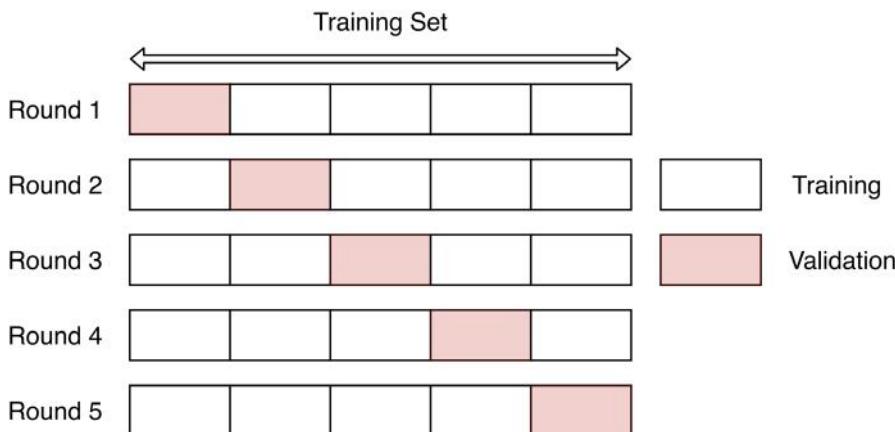


Figure 7.1: Schema of k -fold cross-validation

As you might have already realized, k -fold cross-validation is not really suited for evaluating time series models, as it does not preserve the order of time. For example, in the first round, we train the model using the data from the last 4 folds while evaluating it using the first one.

As k -fold cross-validation is very useful for standard regression and classification tasks, we will come back to it and cover it more in-depth in *Chapter 13, Applied Machine Learning: Identifying Credit Default*.



Bergmeir *et al.* (2018) show that in the case of a purely autoregressive model, the use of standard k -fold cross-validation is possible if the considered models have uncorrelated errors.

Fortunately, we can quite easily adapt the concept of k -fold cross-validation to the time series domain. The resulting approach is called the **walk-forward validation**. In that validation scheme, we expand/slide the training window by one (or multiple) fold(s) at a time.

Figure 7.2 illustrates the expanding window variant of the walk-forward validation, which is also called anchored walk-forward validation. As you can see, we are incrementally increasing the size of the training set, while keeping the next fold as a validation set.

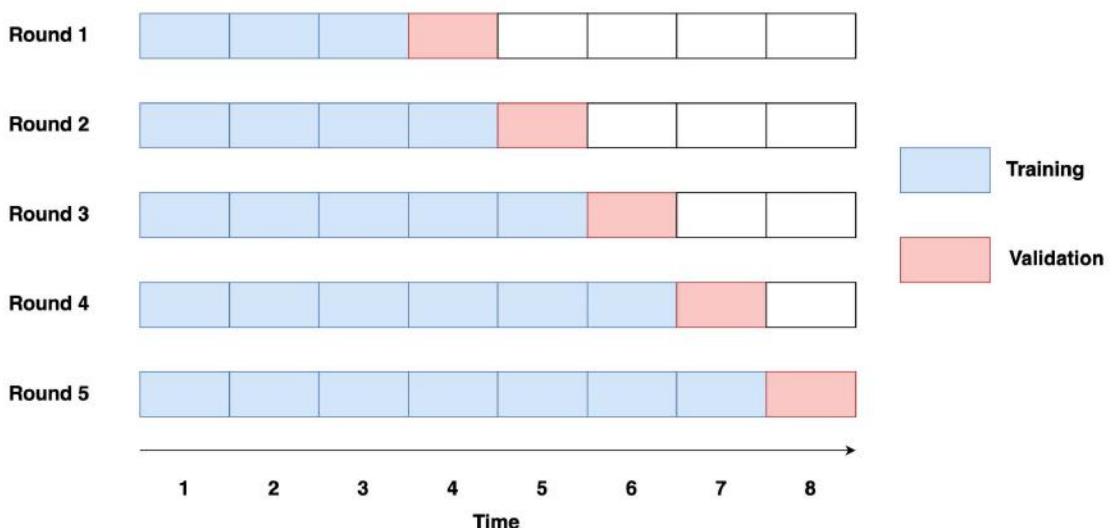


Figure 7.2: Walk-forward validation with an expanding window

This approach comes with a sort of bias—in the earlier rounds, we use much less historical data for training the model than in the latter ones, which makes the errors coming from different rounds not directly comparable. For example, in the first rounds of validation, the model might simply not have enough training data to properly learn the seasonal patterns.

An attempt to solve this problem might be to use a sliding window approach instead of an expanding one. As a result, all models are trained with the same amount of data so the errors are directly comparable. *Figure 7.3* illustrates the process.

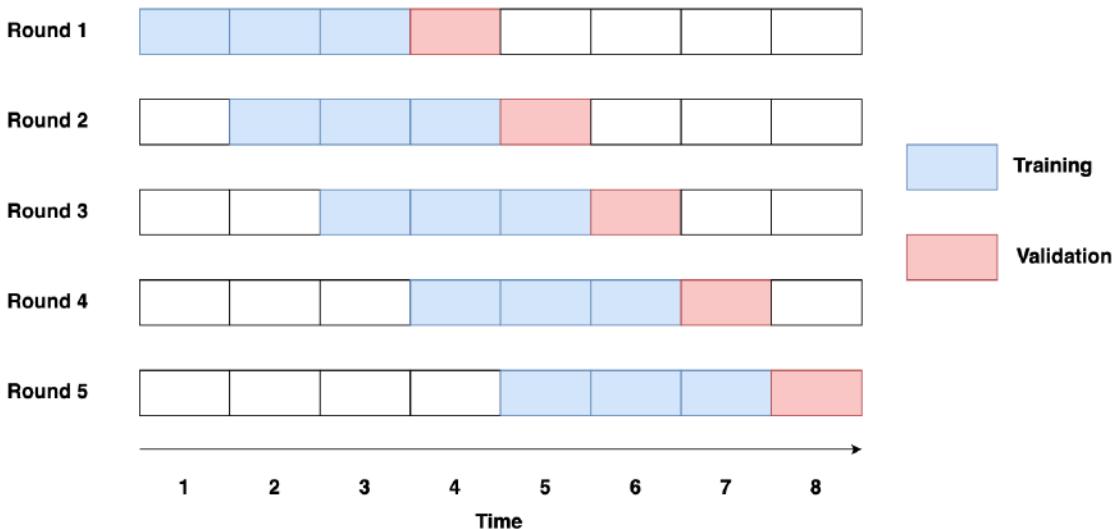


Figure 7.3: Walk-forward validation with a sliding window

We could use this approach when we have a lot of training data (and each sliding window offers enough for the model to learn the patterns well) or when we do not need to look far into the past to learn relevant patterns used to predict the future.



We can use a **nested cross-validation** approach to get even more accurate error estimates while tuning the model's hyperparameters at the same time. In nested CV, there is an outer loop that estimates the model's performance and the inner loop used for hyperparameter tuning. We provide some useful references on the topic in the *See also* section.

In this recipe, we show how to use the walk-forward validation (using both expanding and sliding windows) to evaluate the forecasts of the US unemployment rate.

How to do it...

Execute the following steps to calculate the model's performance using walk-forward validation:

1. Import the libraries and authenticate:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import TimeSeriesSplit, cross_validate
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_percentage_error
import nasdaqdatalink

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

2. Download the monthly US unemployment rate from the years 2010 to 2019:

```
df = (
    nasdaqdatalink.get(dataset="FRED/UNRATENSA",
                        start_date="2010-01-01",
                        end_date="2019-12-31")
    .rename(columns={"Value": "unemp_rate"})
)
df.plot(title="Unemployment rate (US) - monthly")
```

Executing the snippet generates the following plot:

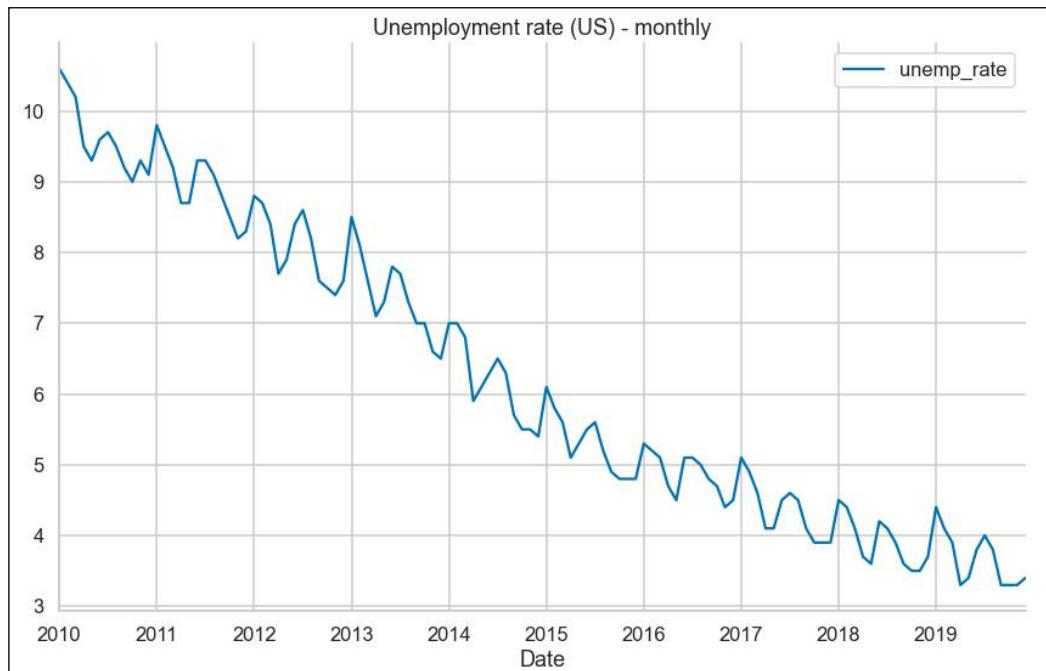


Figure 7.4: Monthly US unemployment rate

3. Create simple features:

```
df["linear_trend"] = range(len(df))
df["month"] = df.index.month
```

As we are avoiding autoregressive features and we know the values of all the features into the future, we are able to forecast for an arbitrarily long forecast horizon.

4. Use one-hot encoding for the month feature:

```
month_dummies = pd.get_dummies(
    df[["month"]], drop_first=True, prefix="month"
)

df = df.join(month_dummies) \
    .drop(columns=[ "month" ])
```

5. Separate the target from the features:

```
X = df.copy()
y = X.pop("unemp_rate")
```

6. Define the expanding window walk-forward validation and print the indices of the folds:

```
expanding_cv = TimeSeriesSplit(n_splits=5, test_size=12)

for fold, (train_ind, valid_ind) in enumerate(expanding_cv.split(X)):
    print(f"Fold {fold} ----")
    print(f"Train indices: {train_ind}")
    print(f"Valid indices: {valid_ind}")
```

Executing the snippet generates the following log:

```
Fold 0 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59]
Valid indices: [60 61 62 63 64 65 66 67 68 69 70 71]
```

```
Fold 1 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71]
Valid indices: [72 73 74 75 76 77 78 79 80 81 82 83]

Fold 2 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83]
Valid indices: [84 85 86 87 88 89 90 91 92 93 94 95]

Fold 3 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83
                 84 85 86 87 88 89 90 91 92 93 94 95]
Valid indices: [96 97 98 99 100 101 102 103 104 105 106 107]

Fold 4 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83
                 84 85 86 87 88 89 90 91 92 93 94 95
                 96 97 98 99 100 101 102 103 104 105 106 107]
Valid indices: [108 109 110 111 112 113 114 115 116 117 118 119]
```

By analyzing the log and keeping in mind that we are working with monthly data, we can see that in the first iteration, the model would be trained using five years of data and evaluated using the sixth year. In the second round, it would be trained using the first six years of data and evaluated using the seventh year, and so on.

- Evaluate the model's performance using the expanding window validation:

```
scores = []

for train_ind, valid_ind in expanding_cv.split(X):
    lr = LinearRegression()
    lr.fit(X.iloc[train_ind], y.iloc[train_ind])
    y_pred = lr.predict(X.iloc[valid_ind])
    scores.append(
        mean_absolute_percentage_error(y.iloc[valid_ind], y_pred)
    )

print(f"Scores: {scores}")
print(f"Avg. score: {np.mean(scores)}")
```

Executing the snippet generates the following output:

```
Scores: [0.03705079312389441, 0.07828415627306308, 0.11981060282173006,
0.16829494012910876, 0.25460459651634165]
Avg. score: 0.1316090177728276
```

The average performance (measured by MAPE) over the cross-validation rounds was 13.2%.

Instead of iterating over the splits, we can easily use the `cross_validate` function from `scikit-learn`:

```
cv_scores = cross_validate(
    LinearRegression(),
    X, y,
    cv=expanding_cv,
    scoring=["neg_mean_absolute_percentage_error",
             "neg_root_mean_squared_error"]
)
pd.DataFrame(cv_scores)
```

Executing the snippet generates the following output:

	fit_time	score_time	test_neg_mean_absolute_percentage_error	test_neg_root_mean_squared_error
0	0.001417	0.000876	-0.037051	-0.232500
1	0.001185	0.000720	-0.078284	-0.433547
2	0.001209	0.000726	-0.119811	-0.520073
3	0.001082	0.000693	-0.168295	-0.662540
4	0.001085	0.000777	-0.254605	-0.928998

Figure 7.5: The scores of each of the validation rounds using a walk-forward CV with an expanding window

By looking at the scores, we see that they are identical (except for the negative sign) to the ones we have obtained by manually iterating over the cross-validation splits.

- Define the sliding window validation and print the indices of the folds:

```
sliding_cv = TimeSeriesSplit(
    n_splits=5, test_size=12, max_train_size=60
)

for fold, (train_ind, valid_ind) in enumerate(sliding_cv.split(X)):
    print(f"Fold {fold} ----")
    print(f"Train indices: {train_ind}")
    print(f"Valid indices: {valid_ind}")
```

Executing the snippet generates the following output:

```
Fold 0 ----
Train indices: [ 0  1  2  3  4  5  6  7  8  9 10 11
                 12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59]
Valid indices: [60 61 62 63 64 65 66 67 68 69 70 71]

Fold 1 ----
Train indices: [12 13 14 15 16 17 18 19 20 21 22 23
                 24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71]
Valid indices: [72 73 74 75 76 77 78 79 80 81 82 83]
```

```

Fold 2 ----
Train indices: [24 25 26 27 28 29 30 31 32 33 34 35
                 36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83]
Valid indices: [84 85 86 87 88 89 90 91 92 93 94 95]

Fold 3 ----
Train indices: [36 37 38 39 40 41 42 43 44 45 46 47
                 48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83
                 84 85 86 87 88 89 90 91 92 93 94 95]
Valid indices: [96 97 98 99 100 101 102 103 104 105 106 107]

Fold 4 ----
Train indices: [48 49 50 51 52 53 54 55 56 57 58 59
                 60 61 62 63 64 65 66 67 68 69 70 71
                 72 73 74 75 76 77 78 79 80 81 82 83
                 84 85 86 87 88 89 90 91 92 93 94 95
                 96 97 98 99 100 101 102 103 104 105 106 107]
Valid indices: [108 109 110 111 112 113 114 115 116 117 118 119]

```

By analyzing the log, we can see the following:

- Each time, the model would be trained using exactly five years of data.
- Between the CV rounds, we are moving by 12 months.
- The validation folds correspond to the ones we saw when we used the expanding window validation. Hence, we can easily compare the scores to see which approach is better.

9. Evaluate the model's performance using the sliding window validation:

```

cv_scores = cross_validate(
    LinearRegression(),
    X, y,
    cv=sliding_cv,
    scoring=["neg_mean_absolute_percentage_error",
             "neg_root_mean_squared_error"]
)
pd.DataFrame(cv_scores)

```

Executing the snippet generates the following output:

	fit_time	score_time	test_neg_mean_absolute_percentage_error	test_neg_root_mean_squared_error
0	0.002245	0.001034	-0.037051	-0.232500
1	0.001400	0.000886	-0.097125	-0.524333
2	0.001330	0.000882	-0.126609	-0.550749
3	0.001364	0.000881	-0.129454	-0.518194
4	0.001397	0.000830	-0.108759	-0.407428

Figure 7.6: The scores of each of the validation rounds using a walk-forward CV with a sliding window

By aggregating the MAPE, we arrive at the average score of 9.98%. It seems that using 5 years of data in each iteration results in a better average score than when using the expanding window. A potential conclusion is that in this particular case, more data does not result in a better model. Instead, we can obtain a better model when using only the most recent data points.

How it works....

First, we imported the required libraries and authenticated with Nasdaq Data Link. In the second step, we downloaded the monthly US unemployment rate. It is the same time series that we worked with in the previous chapter.

In *Step 3*, we created two simple features:

- Linear trend, which is simply the ordinal row number of the ordered time series. Based on the inspection of *Figure 7.4*, we saw that the overall trend in the unemployment rate is decreasing. We hope that this feature will capture that pattern.
- The month index, which identifies from which calendar month the given observation comes.

In *Step 4*, we one-hot encoded the month feature using the `get_dummies` function. We cover one-hot encoding in depth in *Chapter 13, Applied Machine Learning: Identifying Credit Default*, and *Chapter 14, Advanced Concepts for Machine Learning Projects*. In short, we created new columns, each one being a Boolean flag indicating whether the given observation comes from a certain month. Additionally, we dropped the first column to avoid perfect multicollinearity (that is, the infamous dummy variable trap).

In *Step 5*, we separated the features from the target using the `pop` method of a pandas DataFrame.

In *Step 6*, we defined the walk-forward validation using the `TimeSeriesSplit` class from `scikit-learn`. We indicated we want to have 5 splits and that the test size should be 12 months. Ideally, the validation scheme should reflect the real-life usage of the model. In this case, we can state that the ML model will be used to forecast the monthly unemployment rate 12 months into the future.

Then, we used a `for` loop to print the train and validation indices used in each of the cross-validation rounds. The indices returned by the `split` method of the `TimeSeriesSplit` class are ordinal, but we can easily map those to the actual indices of the time series.

We decided not to use autoregressive features, as without them we can forecast arbitrarily long into the future. Naturally, we can also do so with the AR feature, but then we need to handle them appropriately. This specification is simply easier for this use case.

In *Step 7*, we used a very similar `for` loop, this time to evaluate the model's performance. In each iteration of the loop, we trained the linear regression model using that iteration's training data, created predictions for the corresponding validation set, and lastly, calculated the performance expressed as MAPE. We appended the CV scores to a list and then we also calculated the average performance over all 5 rounds of cross-validation.

Instead of using the custom `for` loop, we can use the `cross_validate` function from the `scikit-learn` library. A potential advantage of using it over the loop is that it automatically counts the time spent on the fit and prediction steps of the model. We showed how to obtain the MAPE and MSE scores using this approach.



One thing to note about using the `cross_validate` function (or other `scikit-learn` functionalities such as Grid Search) is that we had to provide the metric names as, for example, "`neg_mean_absolute_percentage_error`". That is the convention used in the `metrics` module of `scikit-learn`, that is, the higher values of the scorers are better than the lower values. Hence, as we want to minimize those metrics, they are negated.

Below, you can find a list of the most popular metrics used for evaluating the accuracy of time series forecasts:

- **Mean Squared Error (MSE)**—One of the most popular metrics in machine learning. As the unit is not very intuitive (not the same unit as the original forecast), we can use MSE to compare the relative performance of various models on the same dataset.
- **Root Mean Squared Error (RMSE)**—By taking the square root of MSE, this metric is now at the same scale as the original time series.
- **Mean Absolute Error (MAE)**—Instead of taking the square, we take the absolute value of the error. As a result, MAE is expressed on the same scale as the original time series. What is more, MAE is more tolerant of outliers, as each observation is given the same weight when calculating the average. In the case of the squared metrics, the outliers were punished more significantly.
- **Mean Absolute Percentage Error (MAPE)**—Very similar to MAE, but expressed as a percentage. Hence, it is easier to understand for many business stakeholders. However, it comes with a serious disadvantage—when the actual value is zero, the metric assumes dividing the error by the actual value, which is not mathematically possible.

Naturally, these are only a few of the selected metrics. It is highly advised to dive deeper into those metrics to fully understand their pros and cons. For example, RMSE is often favored as an optimization metric, as squares are easier to handle than absolute values when mathematical optimization requires taking derivatives.

In *Steps 8* and *9*, we showed how to create the validation scheme using the sliding window approach. The only difference is the fact that we specified the `max_train_size` argument while instantiating the `TimeSeriesSplit` class.



Sometimes we might be interested in creating a gap between the training and validation sets within cross-validation. For example, in the first iteration, the training should be done using the first five values and then the evaluation should be done on the seventh value. We can easily incorporate such a scenario by using the `gap` argument of the `TimeSeriesSplit` class.

There's more...

In this recipe, we have described the standard approach to validating time series models. However, there are many more advanced validation approaches. Actually, most of them come from the financial domain, as validating models based on financial time series proves to be more complex for multiple reasons. We briefly mention some of the more advanced approaches below, together with the challenges they are trying to fix.

One of the limitations of `TimeSeriesSplit` is that it only works at record-level and cannot handle grouping. Imagine we have a dataset of daily stock returns. And due to the specification of our trading algorithm, we are evaluating the performance on a weekly or monthly level and the observations should not overlap between the weekly/monthly groups. *Figure 7.7* illustrates the concept by using the training group size of 3 and validation group size of 1.

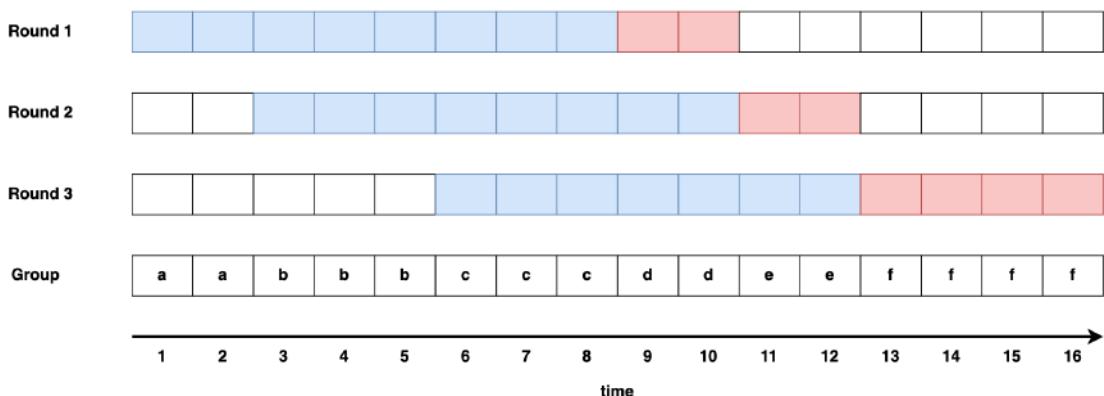


Figure 7.7: Schema of group time series validation

To account for such a grouping of the observations (by week or month), we need to use **group time series validation**, which is a combination of scikit-learn's `TimeSeriesSplit` and `GroupKFold`. There are many implementations of this concept on the internet. One of them can be found in the `m1xtend` library.

To better illustrate the potential problems with forecasting financial time series and evaluating the model's performance, we have to expand our mental model connected to the time series. Such time series actually have two timestamps for each observation:

- A prediction or trade timestamp—when the ML model makes a prediction and we are potentially opening a trade.

- An evaluation or event timestamp—when the response to the prediction/trade becomes available and we can actually calculate the prediction error.

For example, we can have a classification model that predicts the price of certain stock increases or drops by X in the next 5 business days. Based on that prediction, we make a trading decision. We might enter a long position. And over the next 5 days, a lot can happen. The price might or might not move by X , a stop-loss or take-profit mechanism might be triggered, we might just close the position, or any number of possible outcomes. Hence, we can actually evaluate the prediction only at the evaluation timestamp, in this case, after 5 business days.

Such a framework comes with the risk of leaking the information from the test set into the training set. As a result, this is very likely to inflate the model's performance. Hence, we need to make sure that all the data is point-in-time, meaning that is truly available at the time it is used by the model.

For example, near the training/validation split point, there might be training samples whose evaluation time is later than the prediction time of the validation samples. Such overlapping samples are most likely correlated or, in other words, unlikely to be independent, which leads to leaking the information between the sets.

To solve the look-ahead bias, we can apply **purging**. The idea is to drop any samples from the training set whose evaluation time is later than the earliest prediction time of the validation set. In other words, we remove observations whose event time overlaps with the prediction time of the validation set. *Figure 7.8* presents an example.

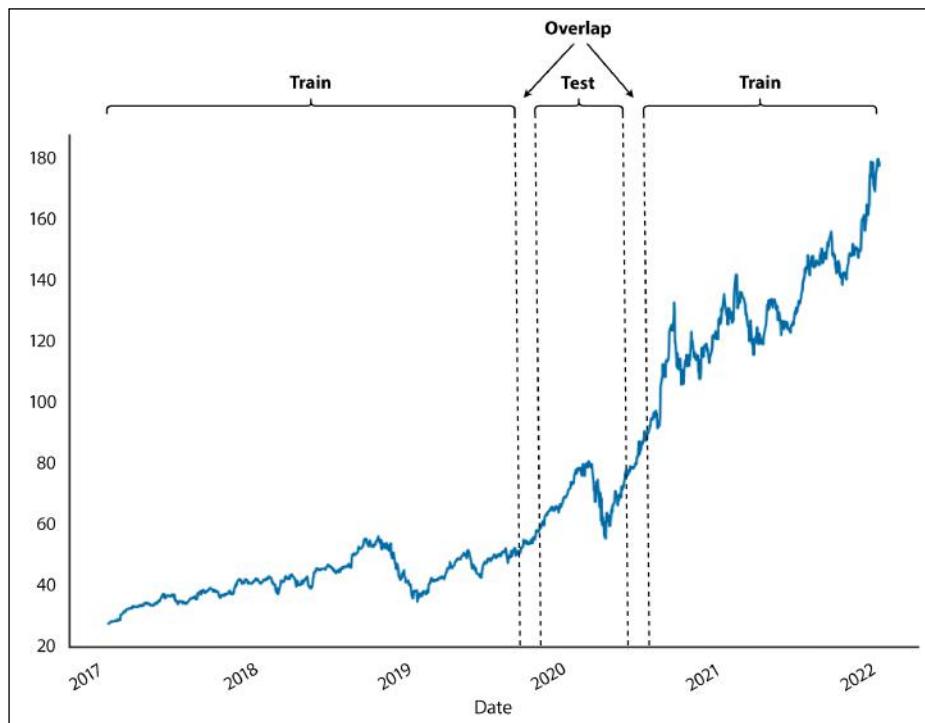


Figure 7.8: Example of purging



You can find the code to run a walk-forward cross-validation with purging in *Advances in financial machine learning* (De Prado, 2018) or in the `timeseriescv` library.

Purging alone might not be sufficient to remove all the leakage, as there might be correlations between the samples over longer periods of time. We can try to solve that by applying an **embargo**, which further eliminates training samples that follow a validation sample. If a training sample's prediction time falls into the embargo period, we simply drop that observation from the train set. We estimate the required size of the embargo period for the problem at hand. *Figure 7.9* illustrates applying both purging and embargo.

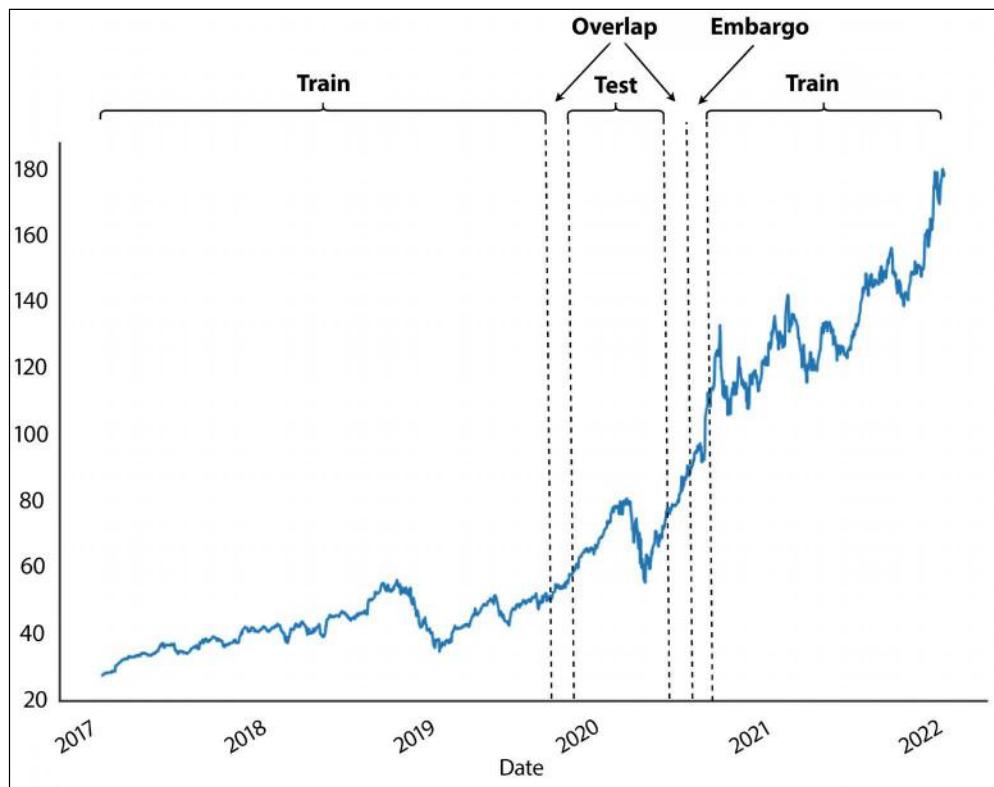


Figure 7.9: Example of purging and embargo

For more details about purging and embargo (as well as their implementation in Python), please refer to *Advances in financial machine learning* (De Prado, 2018).

De Prado (2018) also introduced the **combinatorial purged cross-validation algorithm**, which combines the concepts of purging and embarguing with backtesting (we cover backtesting trading strategies in *Chapter 12, Backtesting Trading Strategies*) and cross-validation.

See also

- Bergmeir, C., & Benítez, J. M. 2012. “On the use of cross-validation for time series predictor evaluation,” *Information Sciences*, 191: 192-213.
- Bergmeir, C., Hyndman, R. J., & Koo, B. 2018. “A note on the validity of cross-validation for evaluating autoregressive time series prediction,” *Computational Statistics & Data Analysis*, 120: 70-83.
- De Prado, M. L. 2018. *Advances in Financial Machine Learning*. John Wiley & Sons.
- Hewamalage, H., Ackermann, K., & Bergmeir, C. 2022. Forecast Evaluation for Data Scientists: Common Pitfalls and Best Practices. *arXiv preprint arXiv:2203.10716*.
- Tashman, L. J. 2000. “Out-of-sample tests of forecasting accuracy: an analysis and review,” *International Journal of Forecasting*, 16(4): 437-450.
- Varma, S., & Simon, R. 2006. “Bias in error estimation when using cross-validation for model selection,” *BMC bioinformatics*, 7(1): 1-8.

Feature engineering for time series

In the previous chapter, we trained some statistical models using just the time series as input. On the other hand, when we want to approach time series forecasting from the ML perspective, **feature engineering** becomes crucial. In the time series context, it means creating informative variables (either from the time series itself or using its timestamp) that help with getting accurate forecasts. Naturally, feature engineering is not only important for the pure ML models but we can use it to enrich the statistical models with external regressors, for example, in the ARIMAX model.

As we have mentioned, there are many ways in which we can create features, and it comes down to a deep understanding of the dataset. Examples of feature engineering include:

- Extracting relevant information from the timestamp. For example, we can extract the year, quarter, month, week number, or day of the week.
- Adding relevant information about special days based on the timestamp. For example, in the retail industry, we might want to add information about all holidays. To get a country-specific holiday calendar, we could use the `holidays` library.
- Adding lagged values of the target, similar to the AR models.
- Creating features based on aggregate values (such as minimum, maximum, mean, median, or standard deviation) over a rolling or expanding window.
- Calculating technical indicators.

In a way, feature generation is only limited by the data, your creativity, or the available time. In this recipe, we show how to create a selection of features based on the timestamp of the time series.

First, we extract the month information and encode it as a dummy variable (one-hot encoding). The biggest issue with this approach in the context of time series is the lack of cyclical continuity in time. It is easiest to understand with an example.

Imagine a scenario of working with energy consumption data. If we use the information about the month of the observed consumption, intuitively it makes sense there should be a connection between two consecutive months, for example, the connection between December and January or between January and February. In comparison, the connection between months further apart, for example, January and July, will probably be weaker. The same logic applies to other time-related information as well, for example, hours within the day.

We present two possible ways of incorporating this information as features. The first one is based on trigonometric functions (sine and cosine transformation). The second one uses radial basis functions to encode similar information.

In this recipe, we work with simulated daily data from the years 2017 to 2019. We chose to simulate the data as the main point of the exercise is to show how different kinds of encoding time information impact the model. And it is easier to show that using simulated data following clear patterns. Naturally, the feature engineering methods shown in this recipe can be applied to any time series.

How to do it...

Execute the following steps to create time-related features and fit linear models using them as inputs:

1. Import the libraries:

```
import numpy as np
import pandas as pd
from datetime import date

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import FunctionTransformer
from sklego.preprocessing import RepeatingBasisFunction
```

2. Generate a time series with repeating patterns:

```
np.random.seed(42)

range_of_dates = pd.date_range(start="2017-01-01",
                               end="2019-12-31")
X = pd.DataFrame(index=range_of_dates)

X["day_nr"] = range(len(X))
X["day_of_year"] = X.index.day_of_year

signal_1 = 2 + 3 * np.sin(X["day_nr"] / 365 * 2 * np.pi)
signal_2 = 2 * np.sin(X["day_nr"] / 365 * 4 * np.pi + 365/2)
noise = np.random.normal(0, 0.81, len(X))

y = signal_1 + signal_2 + noise
```

```
y.name = "y"  
  
y.plot(title="Generated time series")
```

Executing the snippet generates the following plot:

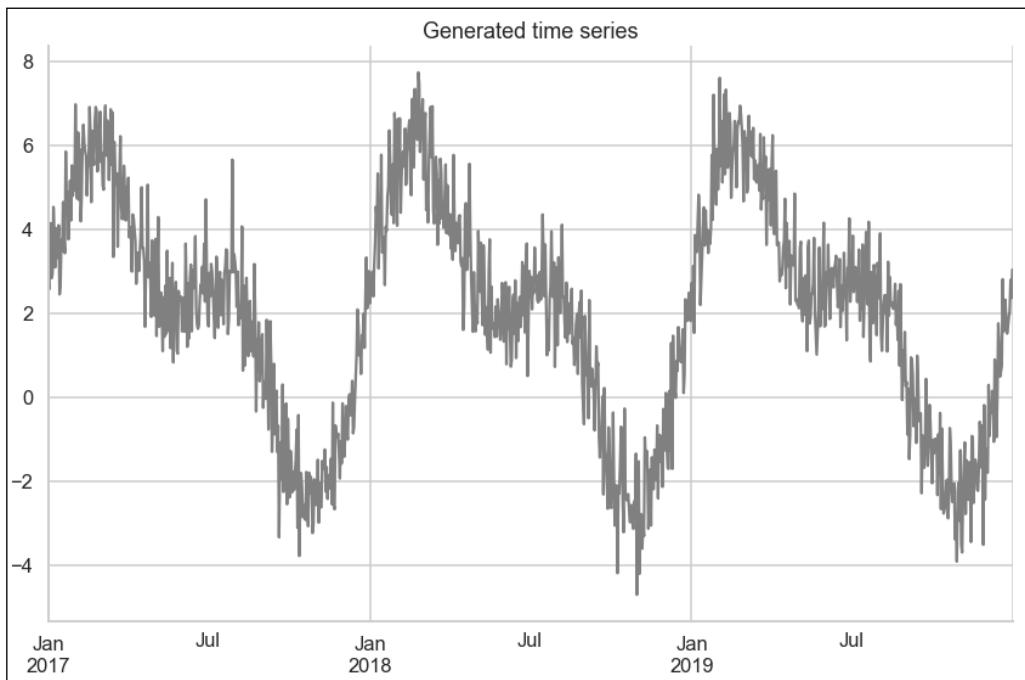


Figure 7.10: The generated time series with repeating patterns

Thanks to the addition of the sine curves and some random noise, we obtained a time series with repeating patterns over the years.

3. Store the time series in a new DataFrame:

```
results_df = y.to_frame()  
results_df.columns = ["y_true"]
```

4. Encode the month information as dummies:

```
X_1 = pd.get_dummies(  
    X.index.month, drop_first=True, prefix="month"  
)  
X_1.index = X.index  
X_1
```

Executing the snippet generates the following preview of the DataFrame with dummy-encoded month features:

	month_2	month_3	month_4	month_5	month_6	month_7	month_8	month_9	month_10	month_11	month_12
2017-01-01	0	0	0	0	0	0	0	0	0	0	0
2017-01-02	0	0	0	0	0	0	0	0	0	0	0
2017-01-03	0	0	0	0	0	0	0	0	0	0	0
2017-01-04	0	0	0	0	0	0	0	0	0	0	0
2017-01-05	0	0	0	0	0	0	0	0	0	0	0
...
2019-12-27	0	0	0	0	0	0	0	0	0	0	1
2019-12-28	0	0	0	0	0	0	0	0	0	0	1
2019-12-29	0	0	0	0	0	0	0	0	0	0	1
2019-12-30	0	0	0	0	0	0	0	0	0	0	1
2019-12-31	0	0	0	0	0	0	0	0	0	0	1

Figure 7.11: Preview of the dummy-encoded month features

5. Fit a linear regression model and plot the in-sample prediction:

```
model_1 = LinearRegression().fit(X_1, y)

results_df["y_pred_1"] = model_1.predict(X_1)
(
    results_df[["y_true", "y_pred_1"]]
    .plot(title="Fit using month dummies")
)
```

Executing the snippet generates the following plot:

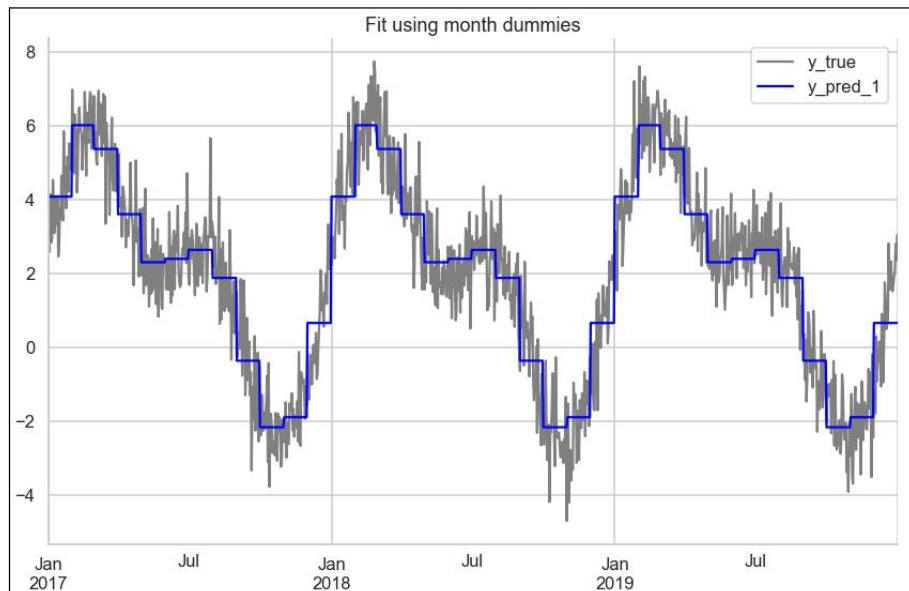


Figure 7.12: The fit obtained using linear regression with the month dummies

We can clearly see the stepwise pattern of the fit, corresponding to 12 unique values of the month feature. The jaggedness of the fit is caused by the discontinuity of the dummy features. With the other approaches, we try to overcome that issue.

6. Define functions used for creating the cyclical encoding:

```
def sin_transformer(period):  
    return FunctionTransformer(lambda x: np.sin(x / period * 2 * np.pi))  
  
def cos_transformer(period):  
    return FunctionTransformer(lambda x: np.cos(x / period * 2 * np.pi))
```

7. Encode the month and day information using cyclical encoding:

```
X_2 = X.copy()  
X_2["month"] = X_2.index.month  
  
X_2["month_sin"] = sin_transformer(12).fit_transform(X_2)[ "month" ]  
X_2["month_cos"] = cos_transformer(12).fit_transform(X_2)[ "month" ]  
  
X_2["day_sin"] = (  
    sin_transformer(365).fit_transform(X_2)[ "day_of_year" ]  
)  
X_2["day_cos"] = (  
    cos_transformer(365).fit_transform(X_2)[ "day_of_year" ]  
)  
  
fig, ax = plt.subplots(2, 1, sharex=True, figsize=(16,8))  
X_2[["month_sin", "month_cos"]].plot(ax=ax[0])  
ax[0].legend(loc="center left", bbox_to_anchor=(1, 0.5))  
X_2[["day_sin", "day_cos"]].plot(ax=ax[1])  
ax[1].legend(loc="center left", bbox_to_anchor=(1, 0.5))  
plt.suptitle("Cyclical encoding with sine/cosine transformation")
```

Executing the snippet generates the following plot:

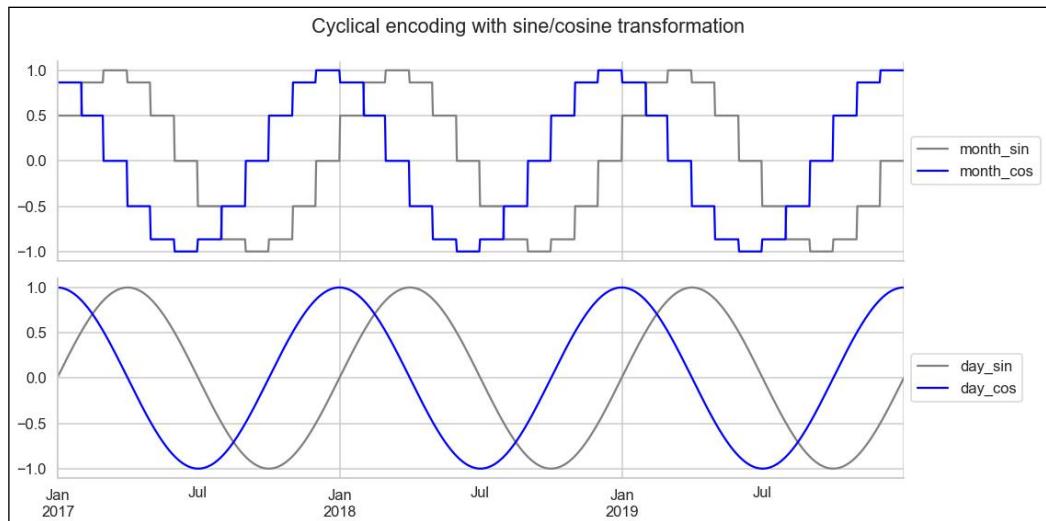


Figure 7.13: Cyclical encoding with sine/cosine transformation

There are two insights we can draw from *Figure 7.13*:

- The curves have a step-wise shape when using the months for encoding. When using daily frequency, the curves are much smoother.
- The plots illustrate the need to use two curves instead of one. As the curves have a repetitive (cyclical) pattern, if we drew a straight horizontal line through the plot for a single year, we would cross the curve in two places. Hence, a single curve would not be enough for the model to understand the observation's time point, as two possibilities exist. Fortunately, with the two curves, there is no such issue.

To clearly see the cyclical representation obtained using this transformation, we can plot the sine and cosine values on a scatterplot for a given year:

```
(  
    X_2[X_2.index.year == 2017]  
    .plot(  
        kind="scatter",  
        x="month_sin",  
        y="month_cos",  
        figsize=(8, 8),  
        title="Cyclical encoding using sine/cosine transformations"  
    )  
)
```

Executing the snippet generates the following plot:

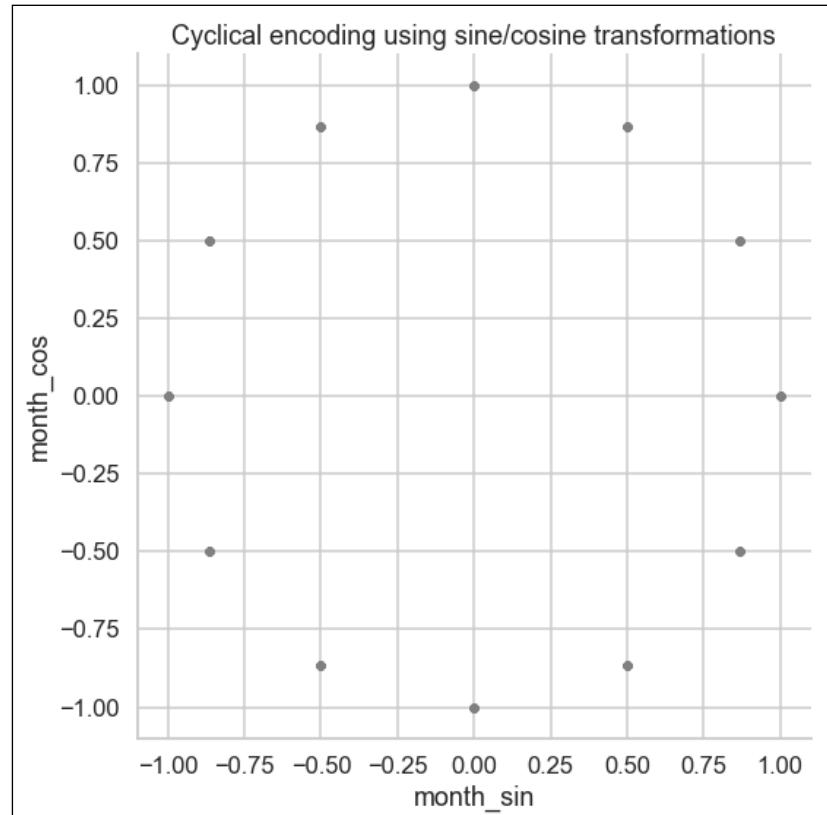


Figure 7.14: The cyclical representation of time

In Figure 7.14, we can see that there are no overlapping values. Hence, the two curves can be used to identify the given observation's point in time.

8. Fit a model using the daily sine/cosine features:

```
X_2 = X_2[["day_sin", "day_cos"]]

model_2 = LinearRegression().fit(X_2, y)

results_df["y_pred_2"] = model_2.predict(X_2)
(
    results_df[["y_true", "y_pred_2"]]
    .plot(title="Fit using sine/cosine features")
)
```

Executing the snippet generates the following plot:

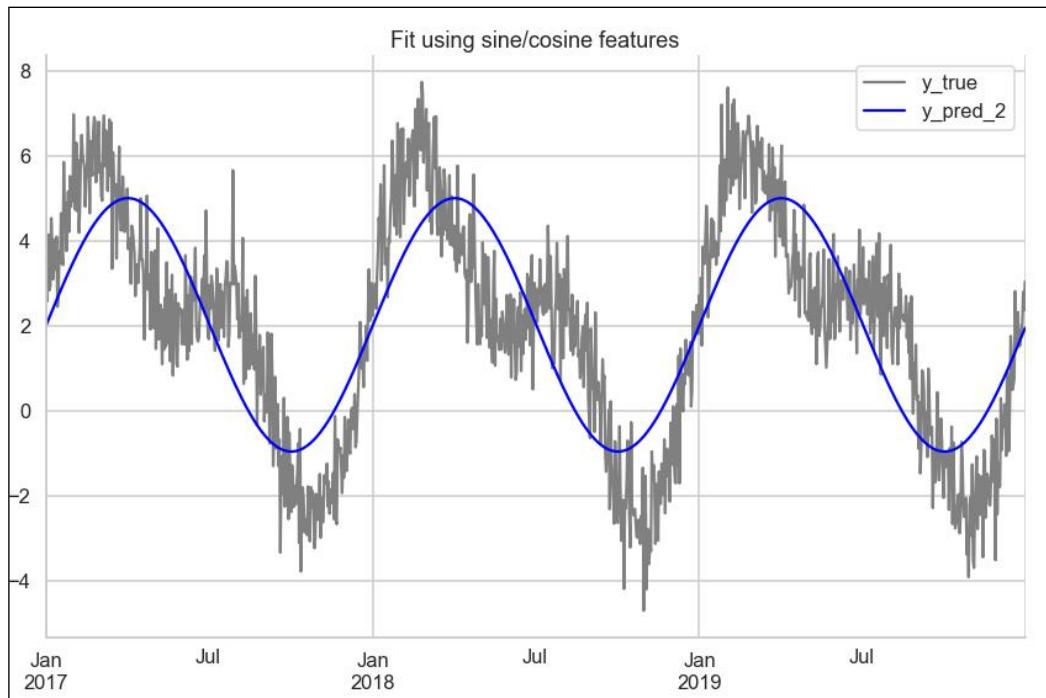


Figure 7.15: The fit obtained using linear regression with the cyclical features

9. Create features using the radial basis functions:

```
rbf = RepeatingBasisFunction(n_periods=12,
                             column="day_of_year",
                             input_range=(1, 365),
                             remainder="drop")
rbf.fit(X)
X_3 = pd.DataFrame(index=X.index,
                     data=rbf.transform(X))

X_3.plot(subplots=True, sharex=True,
          title="Radial Basis Functions",
          legend=False, figsize=(14, 10))
```

Executing the snippet generates the following plot:

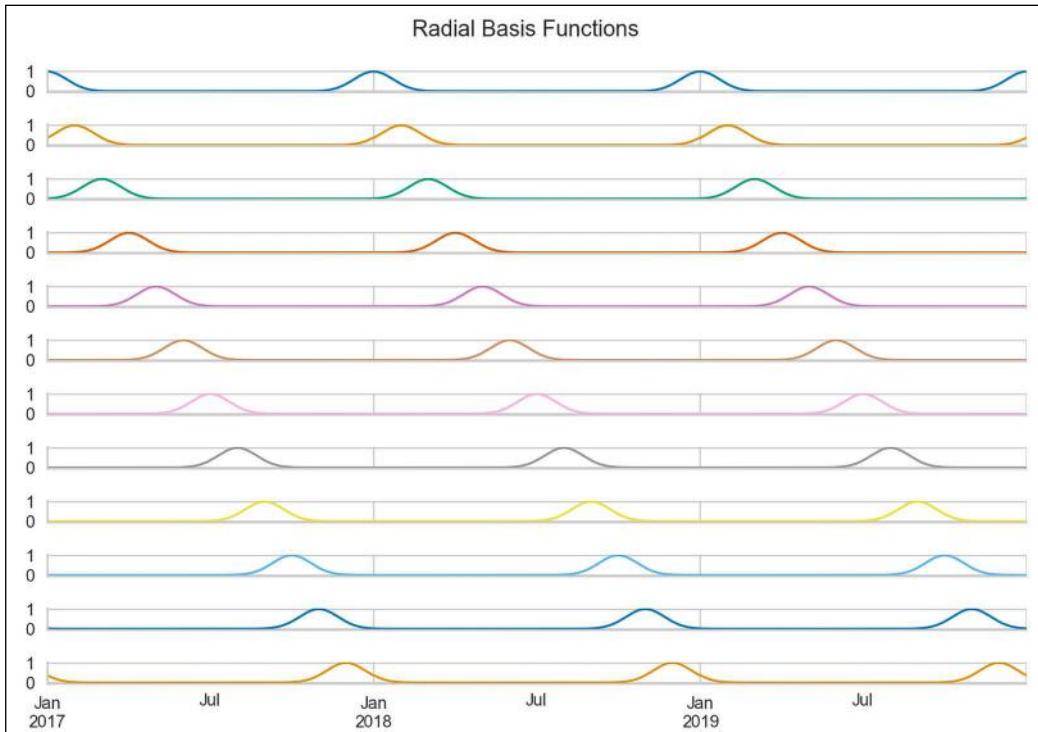


Figure 7.16: Visualization of the features created using the radial basis function

Figure 7.16 presents the 12 curves that we created using the radial basis functions and the day number as input. Each curve tells us how close we are to a certain day of the year. For example, the first curve measures the distance from January 1st. As such, we can observe a peak on the first day of every year, and then it decreases symmetrically as we move away from that date.



The basis functions are equally spaced over the input range. We chose to create 12 curves, as we wanted the radial basis curves to resemble months. This way, each function shows the approximate distance to the first day of the month. The distance is approximate, as the months have unequal lengths.

10. Fit a model using the RBF features:

```
model_3 = LinearRegression().fit(X_3, y)

results_df["y_pred_3"] = model_3.predict(X_3)
(
    results_df[["y_true", "y_pred_3"]]
    .plot(title="Fit using RBF features")
)
```

Executing the snippet generates the following plot:

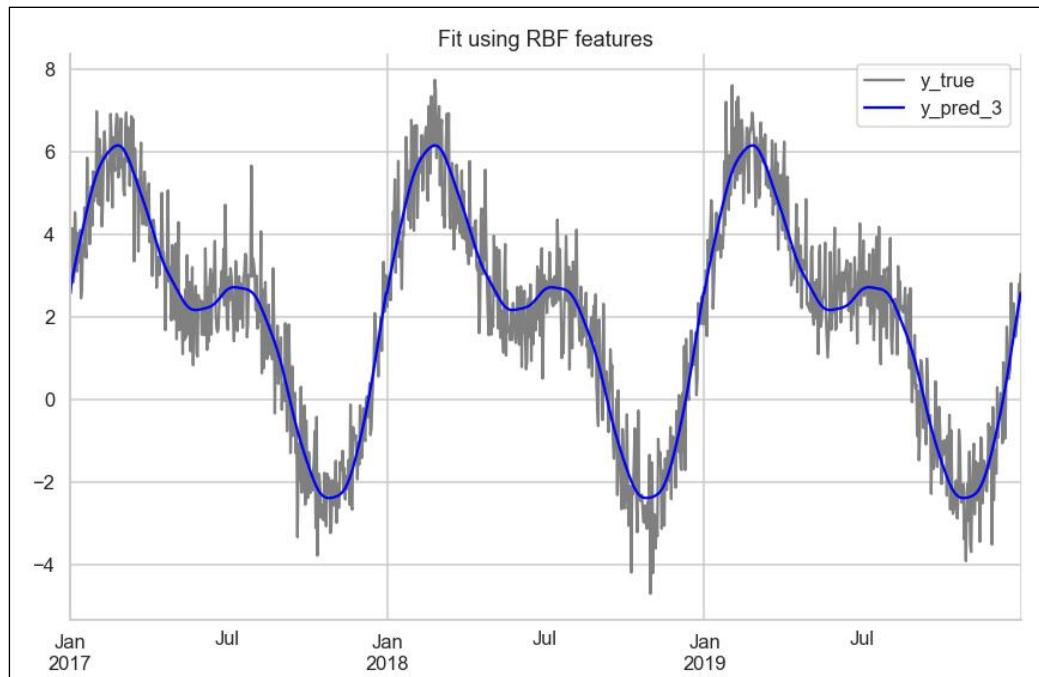


Figure 7.17: The fit obtained using linear regression with the RBF-encoded features

We can clearly see that using the RBF features resulted in the best fit so far.

How it works...

After importing the libraries, we generated the artificial time series by combining two signal lines (created using sine curves) and some random noise. The time series we created spans a period of three years (2017 to 2019). Then, we created two columns for later use:

- `day_nr`—numeric index representing the passage of time. It is equivalent to the ordinal row number.
- `day_of_year`—The ordinal day of the year.

In *Step 3*, we stored the generated time series in a separate DataFrame. We did so in order to store the models' predictions in that DataFrame.

In *Step 4*, we created the month dummies using the `pd.get_dummies` method. For more details on this approach, please refer to the previous recipe.

In *Step 5*, we fitted a linear regression model to the features and used the `predict` method of the fitted model to obtain the fitted values. For predictions, we used the same dataset as we used for training, as we were interested only in the in-sample fit.

In *Step 6*, we defined the functions used for obtaining cyclical encoding with the sine and cosine functions. We created two separate functions, but that is a matter of preference and we could have created a single function to create both features at once. The period argument of the functions corresponds to the number of available periods. For example, when encoding the month number, we would use 12. For the day number, we would use 365 or 366.

In *Step 7*, we encoded both the month and day information using cyclical encoding. We already had the `day_of_year` column with the day number, so we only had to extract the month number from `DatetimeIndex`. Then, we created four columns with cyclical encoding.

In *Step 8*, we dropped all the columns except for the cyclical encoding of the day of the year. Then, we fitted the linear regression model, calculated the fitted values, and plotted the results.



Cyclical encoding has a potentially significant drawback, which is apparent when using tree-based models. By design, tree-based models make a split based on a single feature at the time. And as we have already explained, the sine/cosine features should be considered simultaneously in order to properly identify the time points.

In *Step 9*, we instantiated the `RepeatingBasisFunction` class, which works as a `scikit-learn` transformer. We specified that we wanted 12 RBF curves based on the `day_of_year` column and that the input range is from 1 to 365 (there is no leap year in the sample). Additionally, we specified the `remainder="drop"`, which drops all the other columns that were in the input DataFrame before the transformation. Alternatively, we could have specified the value as "passthrough", which would keep both the old and new features.

It is worth mentioning that there are two key hyperparameters that we can tune when using radial basis functions:

- `n_periods`—The number of the radial basis functions.
- `width`—This hyperparameter is responsible for the shape of the bell curves created with RBFs.

We could use a method such as grid search to identify the optimal values of the hyperparameters for a given dataset. Please refer to *Chapter 13, Applied Machine Learning: Identifying Credit Default*, for more information on the grid search procedure.

In *Step 10*, we once again fitted the model, this time using the RBF features as input.

There's more...

In this recipe, we showed how to manually create time-related features. Naturally, those were just a few of the thousands of possible features we could create. Fortunately, there are Python libraries that facilitate the process of feature engineering/extraction.

We will show two of those. The first approach comes from the `sktime` library, which is a comprehensive library that is the equivalent of `scikit-learn` for time series. The second approach leverages a library called `tsfresh`. The library allows us to automatically generate hundreds or thousands of features with a few lines of code. Under the hood, it uses a combination of established algorithms from statistics, time-series analysis, physics, and signal processing.

We show how to use both approaches in the following steps.

1. Import the libraries:

```
from sktime.transformations.series.date import DateTimeFeatures

from tsfresh import extract_features
from tsfresh.feature_extraction import settings
from tsfresh.utilities.dataframe_functions import roll_time_series
```

2. Extract the datetime features using `sktime`:

```
dt_features = DateTimeFeatures(
    ts_freq="D", feature_scope="comprehensive"
)
features_df_1 = dt_features.fit_transform(y)
features_df_1.head()
```

Executing the snippet generates the following preview of a DataFrame containing the extracted features:

y	year	quarter	month	week_of_year	day	month_of_quarter	week_of_quarter	day_of_quarter	week_of_month	day	weekday
2017-01-01	2.969692	2017	1	1	52	1	1	1	1	1	6
2017-01-02	2.572678	2017	1	1	1	2	1	2	2	1	2
2017-01-03	3.325853	2017	1	1	1	3	1	2	3	1	3
2017-01-04	4.150575	2017	1	1	1	4	1	2	4	1	4
2017-01-05	2.842004	2017	1	1	1	5	1	2	5	1	5

Figure 7.18: Preview of the DataFrame with the extracted features

In the figure, we can see the extracted features. Depending on the ML algorithm we want to use, we might want to further encode those features, for example, using dummy variables.

While instantiating the `DateTimeFeatures` class, we provided the `feature_scope` argument. In this case, we generated a comprehensive set of features. We can also choose the "`minimal`" or "`efficient`" sets.



The extracted features are based on the `DatetimeIndex` of `pandas`. For a comprehensive list of all the features that could be extracted from that index, please refer to the documentation of `pandas`.

3. Prepare the dataset for feature extraction with `tsfresh`:

```
df = y.to_frame().reset_index(drop=False)
df.columns = ["date", "y"]
df["series_id"] = "a"
```

In order to use the feature extraction algorithm, except for the time series itself, our DataFrame must contain columns with a date (or an ordinal encoding of time) and an ID. The latter is required, as the DataFrame might contain multiple time series (in a long format). For example, we could have a DataFrame containing daily stock prices from all the constituents of the S&P 500 index.

4. Create a rolled-up DataFrame for feature extraction:

```
df_rolled = roll_time_series(
    df, column_id="series_id", column_sort="date",
    max_timeshift=30, min_timeshift=7
).drop(columns=["series_id"])
df_rolled
```

Executing the snippet generates the following preview of a rolled-up DataFrame:

	date	y	id
3410	2017-01-01	2.969692	(a, 2017-01-08 00:00:00)
3411	2017-01-02	2.572678	(a, 2017-01-08 00:00:00)
3412	2017-01-03	3.325853	(a, 2017-01-08 00:00:00)
3413	2017-01-04	4.150575	(a, 2017-01-08 00:00:00)
3414	2017-01-05	2.842004	(a, 2017-01-08 00:00:00)
...
33447	2019-12-27	1.914564	(a, 2019-12-31 00:00:00)
33448	2019-12-28	2.062146	(a, 2019-12-31 00:00:00)
33449	2019-12-29	2.801118	(a, 2019-12-31 00:00:00)
33450	2019-12-30	2.372868	(a, 2019-12-31 00:00:00)
33451	2019-12-31	3.042800	(a, 2019-12-31 00:00:00)

Figure 7.19: Preview of a rolled-up DataFrame

We used a sliding window to roll up the DataFrame because we wanted to achieve the following:

- Calculate meaningful aggregate features for time series forecasting. For example, we might calculate the min/max values in the last 10 days, or the 20-day Simple Moving Average technical indicator. Each time, those calculations involve a time window, as calculating those aggregate measures using one observation would simply make no sense.
- Extract the features for all available time points, so we can easily plug them into our ML forecasting model. This way, we are basically creating the entire training dataset at once.

To do so, we used the `roll_time_series` function to create a rolled-up DataFrame, which will be then used for feature extraction. We specified the minimum and maximum window sizes. In our case, we will discard windows shorter than 7 days and we will use a maximum of 30 days.

In *Figure 7.19*, we can see the newly added `id` column. As we can see, multiple observations have the same values in the `id` column. For example, the value of `(a, 2017-01-08 00:00:00)` indicates that we are using that particular data point when extracting the features from the time series labeled as `a` (we created this ID artificially in the previous step) for the time point that includes the last 30 days until 2017-01-08. Having prepared the rolled-up DataFrame, we can extract the features.

5. Extract the minimal set of features:

```
settings_minimal = settings.MinimalFCParameters()  
settings_minimal
```

Executing the snippet generates the following output:

```
{'sum_values': None,  
 'median': None,  
 'mean': None,  
 'length': None,  
 'standard_deviation': None,  
 'variance': None,  
 'maximum': None,  
 'minimum': None}
```

In the dictionary, we can see all the features that will be created. The `None` value implies that the feature has no additional hyperparameters. We chose to extract the minimum set, as the other ones would take a significant amount of time. Alternatively, we could use `settings.EfficientFCParameters` or `settings.ComprehensiveFCParameters` to generate hundreds or thousands of features.

With the following snippet, we actually extract the features:

```
features_df_2 = extract_features(  
    df_rolled, column_id="id",  
    column_sort="date",  
    default_fc_parameters=settings_minimal  
)
```

6. Clean up the index and inspect the features:

```
features_df_2 = (
    features_df_2
    .set_index(
        features_df_2.index.map(lambda x: x[1]), drop=True
    )
)
features_df_2.index.name = "last_date"
features_df_2.head(25)
```

Executing the snippet generates the following output:

last_date	y_sum_values	y_median	y_mean	y_length	y_standard_deviation	y_variance	y_maximum	y_minimum
2017-01-08	27.3446	3.1478	3.4181	8.0	0.6696	0.4484	4.5371	2.5727
2017-01-09	30.4437	3.0991	3.3826	9.0	0.6392	0.4086	4.5371	2.5727
2017-01-10	34.4712	3.2125	3.4471	10.0	0.6365	0.4052	4.5371	2.5727
2017-01-11	37.7911	3.3199	3.4356	11.0	0.6080	0.3697	4.5371	2.5727
2017-01-12	41.2148	3.3229	3.4346	12.0	0.5821	0.3389	4.5371	2.5727
2017-01-13	45.3157	3.3259	3.4858	13.0	0.5868	0.3443	4.5371	2.5727
2017-01-14	47.7731	3.3229	3.4124	14.0	0.6244	0.3899	4.5371	2.4575
2017-01-15	50.4837	3.3199	3.3656	15.0	0.6281	0.3945	4.5371	2.4575
2017-01-16	54.2347	3.3229	3.3897	16.0	0.6153	0.3786	4.5371	2.4575
2017-01-17	57.7174	3.3259	3.3951	17.0	0.5973	0.3568	4.5371	2.4575
2017-01-18	62.3698	3.3748	3.4650	18.0	0.6480	0.4199	4.6524	2.4575
2017-01-19	66.1248	3.4237	3.4803	19.0	0.6340	0.4020	4.6524	2.4575
2017-01-20	69.5618	3.4304	3.4781	20.0	0.6180	0.3820	4.6524	2.4575
2017-01-21	75.4183	3.4370	3.5913	21.0	0.7876	0.6203	5.8565	2.4575
2017-01-22	79.9908	3.4599	3.6359	22.0	0.7962	0.6339	5.8565	2.4575
2017-01-23	84.8846	3.4827	3.6906	23.0	0.8198	0.6721	5.8565	2.4575
2017-01-24	88.6509	3.6168	3.6938	24.0	0.8027	0.6444	5.8565	2.4575
2017-01-25	93.2093	3.7510	3.7284	25.0	0.8045	0.6473	5.8565	2.4575
2017-01-26	98.3749	3.7530	3.7836	26.0	0.8359	0.6988	5.8565	2.4575
2017-01-27	102.5922	3.7550	3.7997	27.0	0.8244	0.6796	5.8565	2.4575
2017-01-28	108.1175	3.7607	3.8613	28.0	0.8706	0.7579	5.8565	2.4575
2017-01-29	112.9206	3.7664	3.8938	29.0	0.8725	0.7613	5.8565	2.4575
2017-01-30	118.0400	3.8787	3.9347	30.0	0.8856	0.7843	5.8565	2.4575
2017-01-31	122.9716	3.9909	3.9668	31.0	0.8888	0.7900	5.8565	2.4575
2017-02-01	126.9819	4.0275	4.0962	31.0	1.0169	1.0341	6.9799	2.4575

Figure 7.20: Preview of the features generated with tsfresh

In Figure 7.20, we can see that the minimum window length is 8, while the maximum one is 31. That is as intended, as we indicated we wanted to use the minimum size of 7, which translates to 7 prior days plus the current one. Similarly for the maximum value.



sktime also offers a wrapper around tsfresh. We can access the feature generation algorithm by using sktime's TSFreshFeatureExtractor class.

It is also worth mentioning that `tsfresh` has three other very interesting features:

- A feature selection algorithm based on hypothesis tests. As the library is capable of generating hundreds or thousands of features, it is definitely important to select the ones that are relevant to our use case. To do so, the library uses the *fresh* algorithm, which stands for *feature extraction based on scalable hypothesis tests*.
- The ability to handle feature generation and selection for large datasets by employing parallel processing with either multiprocessing on a local machine or using Spark or Dask clusters when the data does not fit into a single machine.
- It offers transformer classes (for example, `FeatureAugmenter` or `FeatureSelector`), which we can use together with `scikit-learn` pipelines. We cover pipelines in *Chapter 13, Applied Machine Learning: Identifying Credit Default*.



`tsfresh` is only one of the available libraries for automatic feature generation for time series data. Other libraries include `feature_engine` and `tsflex`.

Time series forecasting as reduced regression

Until now, we have mostly used dedicated time series models for forecasting tasks. On the other hand, it would also be interesting to experiment with other algorithms that are typically used for solving regression tasks. This way, we might improve the performance of our models.

One of the reasons to use those models is their flexibility. For example, we could go beyond univariate setup, that is, we could enrich our dataset with a wide variety of additional features. We have covered some approaches to feature engineering in the previous recipe. Alternatively, we could add external regressors such as time series, which historically proved to be correlated with the target of our forecasting exercise.



When adding additional time series as external regressors, we should be cautious about their availability. If we do not know their future values, we might use their lagged values or forecast them separately and feed them back into the initial model.

Given the temporal dependency of the time series data (relevant for the lagged values of the time series), we cannot directly use regression models for time series forecasting. First, we need to convert such temporal data into a supervised learning problem, to which we can apply traditional regression algorithms. That process is called **reduction** and it decomposes certain learning tasks (time series forecasting) into simpler tasks. Then, those can be composed again to offer a solution to the original task. In other words, reduction refers to the concept of using an algorithm or model to solve a learning task that it was not originally designed for. Hence, in **reduced regression**, we are effectively transforming a forecasting task into a tabular regression problem.

In practice, reduction uses a sliding window to split the time series into fixed-length windows. It will be easier to understand how reduction works with an example. Imagine a time series of consecutive numbers from 1 to 100. Then, we take a sliding window of length 5. The first window contains observations 1 to 4 as features and observation 5 as the target. The second window uses observations 2 to 5 as features and observation 6 as the target. And so on. Once we arrange all those windows on top of each other, we obtain a tabular format of the data that allows us to use traditional regression algorithms for time series forecasting. *Figure 7.21* illustrates the reduction procedure.

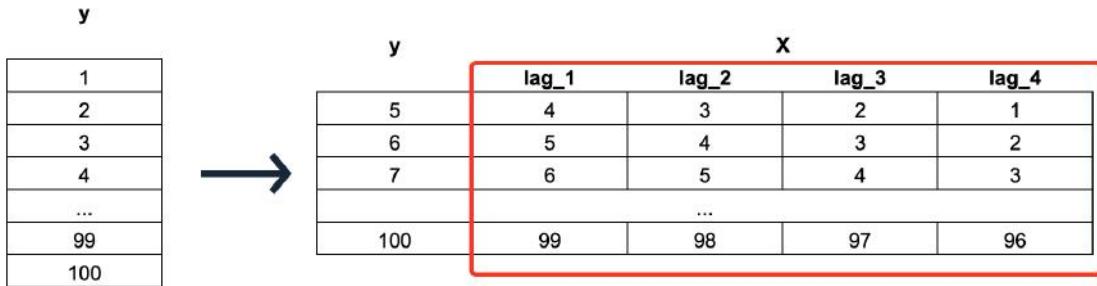


Figure 7.21: Schema of the reduction procedure

It is also worth mentioning that there are some nuances to working with reduced regression. For example, reduced regression models lose the typical characteristics of time series models, that is, they lose the notion of time. As a result, they are unable to handle trends and seasonality. That is why it is often useful to first detrend and deseasonalize the data and only then perform the reduction. Intuitively, this is similar to modeling only the AR terms. Deseasonalizing and detrending the data first makes it easier to find a better fitting model as we are not accounting for trend and seasonality on top of the AR terms.

In this recipe, we show an example of a reduced regression procedure using the US unemployment rates dataset.

Getting ready

In this recipe, we are working with the already familiar US unemployment rates time series. For brevity, we do not repeat the steps on how to download the data. You can find the code in the accompanying notebook. For the remainder of the recipe, assume that the downloaded data is in a DataFrame called `y`.

How to do it...

Execute the following steps to create 12 steps ahead forecasts of the US unemployment rate using reduced regression:

1. Import the libraries:

```
from sktime.utils.plotting import plot_series
from sktime.forecasting.model_selection import (
    temporal_train_test_split, ExpandingWindowSplitter
)
```

```

from sktime.forecasting.base import ForecastingHorizon
from sktime.forecasting.compose import (
    make_reduction, TransformedTargetForecaster, EnsembleForecaster
)
from sktime.performance_metrics.forecasting import (
    mean_absolute_percentage_error
)
from sktime.transformations.series.detrend import (
    Deseasonalizer, Detrender
)
from sktime.forecasting.trend import PolynomialTrendForecaster
from sktime.forecasting.model_evaluation import evaluate
from sktime.forecasting.arima import AutoARIMA
from sklearn.ensemble import RandomForestRegressor

```

2. Split the time series into training and tests sets:

```

y_train, y_test = temporal_train_test_split(
    y, test_size=12
)
plot_series(
    y_train, y_test,
    labels=["y_train", "y_test"]
)

```

Executing the snippet generates the following plot:



Figure 7.22: The time series divided into training and test sets

3. Set the forecast horizon to 12 months:

```

fh = ForecastingHorizon(y_test.index, is_relative=False)
fh

```

Executing the snippet generates the following output:

```

ForecastingHorizon(['2019-01', '2019-02', '2019-03', '2019-04', '2019-05',
'2019-06', '2019-07', '2019-08', '2019-09', '2019-10', '2019-11',
'2019-12'], dtype='period[M]', is_relative=False)

```

Whenever we will use this `fh` object to create forecasts, we will create forecasts for the 12 months of 2019.

4. Instantiate the reduced regression model, fit it to the data, and create predictions:

```
regressor = RandomForestRegressor(random_state=42)
rf_forecaster = make_reduction(
    estimator=regressor,
    strategy="recursive",
    window_length=12
)
rf_forecaster.fit(y_train)
y_pred_1 = rf_forecaster.predict(fh)
```

5. Evaluate the performance of the forecasts:

```
mape_1 = mean_absolute_percentage_error(
    y_test, y_pred_1, symmetric=False
)
fig, ax = plot_series(
    y_train["2016":], y_test, y_pred_1,
    labels=["y_train", "y_test", "y_pred"]
)
ax.set_title(f"MAPE: {100*mape_1:.2f}%")
```

Executing the snippet generates the following plot:

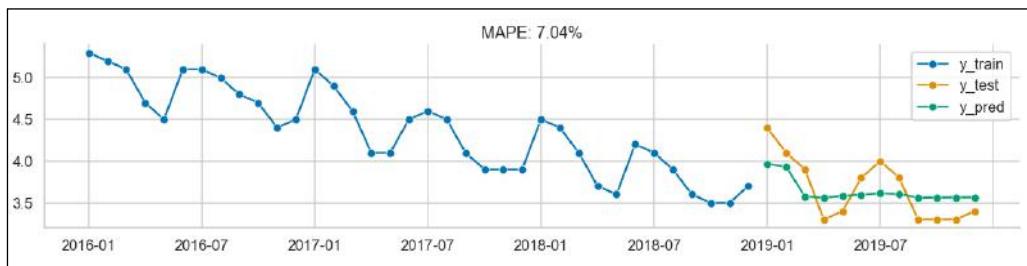


Figure 7.23: Forecasts vs. actuals using the reduced Random Forest

The almost flat forecast is most likely connected to the drawback of the reduced regression approach we have mentioned in the introduction. By reshaping the data into a tabular format, we are effectively losing information about trends and seasonality. To account for those, we can first deseasonalize and detrend the time series and only then use the reduced regression approach.

6. Deseasonalize the time series:

```
deseasonalizer = Deseasonalizer(model="additive", sp=12)
y_deseas = deseasonalizer.fit_transform(y_train)
plot_series(
    y_train, y_deseas,
    labels=["y_train", "y_deseas"]
)
```

Executing the snippet generates the following plot:

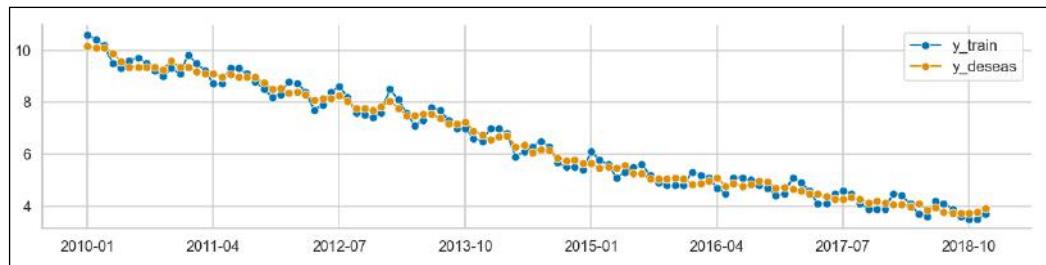


Figure 7.24: The original time series and the deseasonalized one

To provide more context, we can plot the extracted seasonal component:

```
plot_series(
    deseasonalizer.seasonal_,
    labels=["seasonal_component"]
)
```

Executing the snippet generates the following plot:

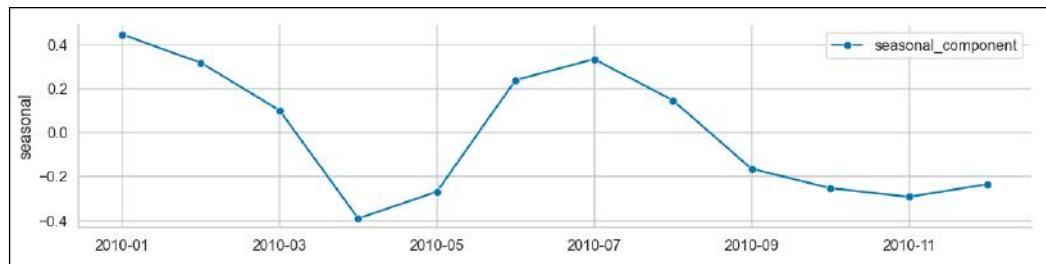


Figure 7.25: The extracted seasonal component

While analyzing Figure 7.25, we should not pay much attention to the x-axis labels, as the extracted seasonal pattern is the same for each year.

7. Detrend the time series:

```

forecaster = PolynomialTrendForecaster(degree=1)
transformer = Detrender(forecaster=forecaster)
y_detrend = transformer.fit_transform(y_deseas)

# in-sample predictions
forecaster = PolynomialTrendForecaster(degree=1)
y_in_sample = (
    forecaster
    .fit(y_deseas)
    .predict(fh=-np.arange(len(y_deseas)))
)

plot_series(
    y_deseas, y_in_sample, y_detrend,
    labels=["y_deseas", "linear trend", "resids"]
)

```

Executing the snippet generates the following plot:

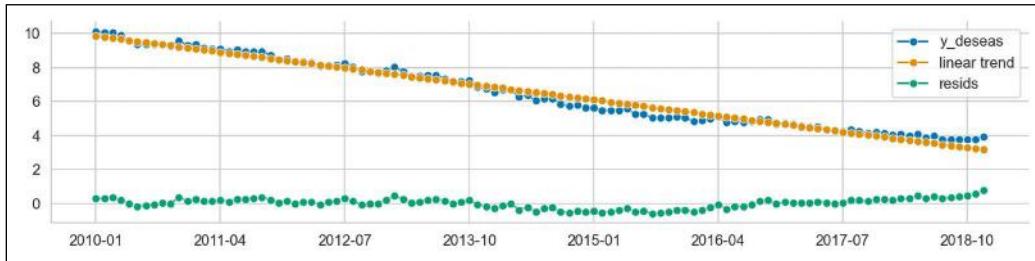


Figure 7.26: The deseasonalized time series together with the fitted linear trend and the corresponding residuals

In *Figure 7.26*, we can see 3 lines:

- The deseasonalized time series from the previous step
- The linear trend fitted to the deseasonalized time series
- The residuals, which are created by subtracting the fitted linear trend from the deseasonalized time series

8. Combine the components into a pipeline, fit it to the original time series, and obtain predictions:

```
rf_pipe = TransformedTargetForecaster(
    steps = [
        ("deseasonalize", Deseasonalizer(model="additive", sp=12)),
        ("detrend", Detrender(
            forecaster=PolynomialTrendForecaster(degree=1)
        )),
        ("forecast", rf_forecaster),
    ]
)
rf_pipe.fit(y_train)
y_pred_2 = rf_pipe.predict(fh)
```

9. Evaluate the pipeline's predictions:

```
mape_2 = mean_absolute_percentage_error(
    y_test, y_pred_2, symmetric=False
)
fig, ax = plot_series(
    y_train["2016":], y_test, y_pred_2,
    labels=["y_train", "y_test", "y_pred"]
)
ax.set_title(f"MAPE: {100*mape_2:.2f}%")
```

Executing the snippet generates the following plot:

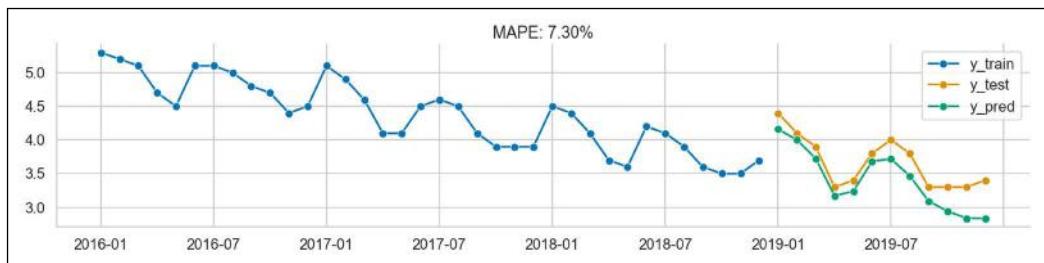


Figure 7.27: The fit of the pipeline containing deseasonalization and detrending before reduced regression

By analyzing *Figure 7.27*, we can draw the following conclusions:

- The shape of the forecast obtained using the pipeline is much more similar to the actual values—it captures the trend and seasonality components.
- The error measured by MAPE seems to be worse than that of an almost flat line forecast visible in *Figure 7.23*.

10. Evaluate the performance using expanding window cross-validation:

```

cv = ExpandingWindowSplitter(
    fh=list(range(1,13)),
    initial_window=12*5,
    step_length=12
)

cv_df = evaluate(
    forecaster=rf_pipe,
    y=y,
    cv=cv,
    strategy="refit",
    return_data=True
)

cv_df

```

Executing the snippet generates the following DataFrame:

	test_MeanAbsolutePercentageError	fit_time	pred_time	len_train_window	cutoff	y_train	y_test	y_pred
0	0.017968	0.060766	0.032131	60	2014-12	unemp_rate 2010-01 10.6 2010-0...	unemp_rate 2015-01 6.1 2015-0...	unemp_rate 2015-01 6.189424 2015-0...
1	0.072160	0.058252	0.032728	72	2015-12	unemp_rate 2010-01 10.6 2010-0...	unemp_rate 2016-01 5.3 2016-0...	unemp_rate 2016-01 5.424690 2016-0...
2	0.092562	0.058367	0.032157	84	2016-12	unemp_rate 2010-01 10.6 2010-0...	unemp_rate 2017-01 5.1 2017-0...	unemp_rate 2017-01 5.043032 2017-0...
3	0.098999	0.060876	0.031808	96	2017-12	unemp_rate 2010-01 10.6 2010-0...	unemp_rate 2018-01 4.5 2018-0...	unemp_rate 2018-01 4.380789 2018-0...
4	0.072970	0.064576	0.032254	108	2018-12	unemp_rate 2010-01 10.6 2010-0...	unemp_rate 2019-01 4.4 2019-0...	unemp_rate 2019-01 4.161463 2019-0...

Figure 7.28: The DataFrame containing the cross-validation results

Additionally, we can investigate the range of dates used for training and evaluating the pipeline within the cross-validation procedure:

```

for ind, row in cv_df.iterrows():
    print(f"Fold {ind} ----")
    print(f"Training: {row['y_train'].index.min()} - {row['y_train'].index.max()}")
    print(f"Training: {row['y_test'].index.min()} - {row['y_test'].index.max()}")

```

Executing the snippet generates the following output:

```

Fold 0 ----
Training: 2010-01 - 2014-12
Training: 2015-01 - 2015-12
Fold 1 ----
Training: 2010-01 - 2015-12
Training: 2016-01 - 2016-12
Fold 2 ----

```

```

Training: 2010-01 - 2016-12
Training: 2017-01 - 2017-12
Fold 3 ----
Training: 2010-01 - 2017-12
Training: 2018-01 - 2018-12
Fold 4 ----
Training: 2010-01 - 2018-12
Training: 2019-01 - 2019-12

```

Effectively, we have created a 5-fold cross-validation in which the expanding window is growing by 12 months between the folds and we are always evaluating using the following 12 months.

11. Plot the predictions from the cross-validation folds:

```

n_fold = len(cv_df)

plot_series(
    y,
    *[cv_df["y_pred"].iloc[x] for x in range(n_fold)],
    markers=["o", *["."] * n_fold],
    labels=["y_true"] + [f"cv: {x}" for x in range(n_fold)]
)

```

Executing the snippet generates the following plot:

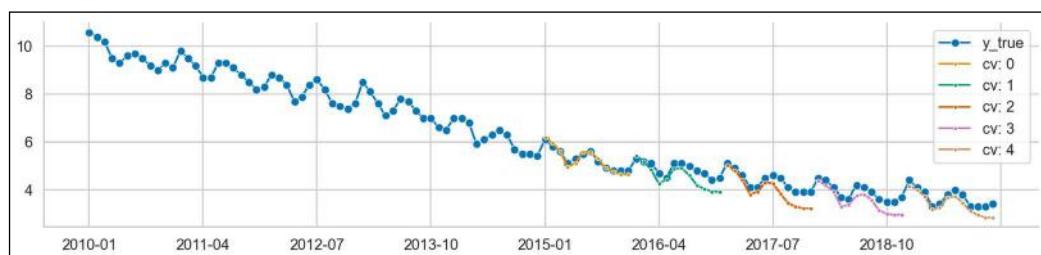


Figure 7.29: Forecasts from each of the cross-validation folds plotted against the actuals

12. Create an ensemble forecast using the RF pipeline and AutoARIMA:

```

ensemble = EnsembleForecaster(
    forecasters = [
        ("autoarima", AutoARIMA(sp=12)),
        ("rf_pipe", rf_pipe)
    ]
)
ensemble.fit(y_train)
y_pred_3 = ensemble.predict(fh)

```

In this case, we fitted an AutoARIMA model directly to the original time series. However, we could have also deseasonalized and detrended the time series before fitting the model. In such a scenario, indicating the seasonal period might not have been necessary (depending on how well the seasonality is removed using classical decomposition).

13. Evaluate the ensemble's predictions:

```
mape_3 = mean_absolute_percentage_error(
    y_test, y_pred_3, symmetric=False
)
fig, ax = plot_series(
    y_train["2016":], y_test, y_pred_3,
    labels=["y_train", "y_test", "y_pred"]
)
ax.set_title(f"MAPE: {100*mape_3:.2f}%")
```

Executing the snippet generates the following plot:

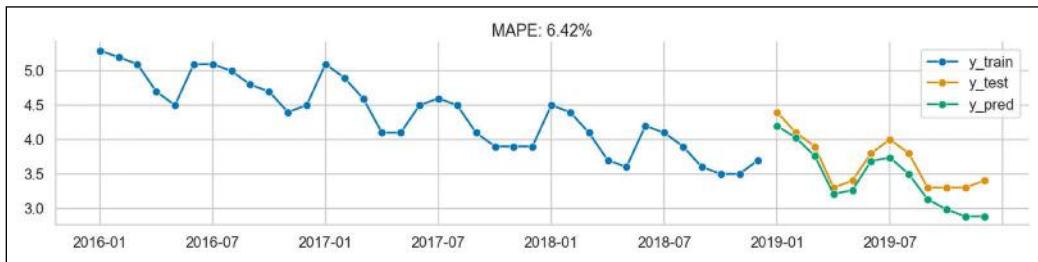


Figure 7.30: The fit of the ensemble model aggregating the reduced regression pipeline and AutoARIMA

As we can see in *Figure 7.30*, ensembling the two models results in improved performance compared to the reduced Random Forest pipeline.

How it works...

After importing the libraries, we used the `temporal_train_test_split` function to split the data into training and test sets. We kept the last 12 observations (the entire 2019) as a test set. We also plotted the time series using the `plot_series` function, which is especially useful when we want to plot multiple time series in a single plot.

In *Step 3*, we defined the `ForecastingHorizon`. In `sktime`, the forecasting horizon can be an array of values that are either relative (indicating time differences compared to the latest time point in the training data) or absolute (indicating specific points in time). In our case, we used the absolute values by providing the indices of the test set and setting `is_relative=False`.



On the other hand, the relative values of the forecasting horizon include a list of steps for which we want to obtain predictions. The relative horizon could be very useful when making rolling predictions, as we can reuse it when we add new data.

In *Step 4*, we fitted a reduced regression model to the training data. To do so, we used the `make_reduction` function and provided three arguments. The `estimator` argument is used to indicate any regression model that we would like to use in the reduced regression setting. In this case, we chose Random Forest (more details on the Random Forest algorithm can be found in *Chapter 14, Advanced Concepts for Machine Learning Projects*). The `window_length` indicates how many past observations to use to create the reduced regression task, that is, convert the time series into a tabular dataset. Lastly, the `strategy` argument determines the way multi-step forecasts will be created. We can choose one of the following strategies to obtain multi-step forecasts:

- **Direct**—This strategy assumes creating a separate model for each horizon we are forecasting. In our case, we are forecasting 12 steps ahead. This would mean that the strategy would create 12 separate models to obtain the forecasts.
- **Recursive**—This strategy assumes fitting a single one-step ahead model. However, to create the forecasts, it uses the previous time step's output as the input for the next time step. For example, to obtain the forecast for the second observation into the future, it would use the forecast obtained for the first observation into the future as part of the feature set.
- **Multiooutput**—In this strategy, we use one model to predict all the values for the entire forecast horizon. This strategy depends on having a model capable of predicting entire sequences in one go.

After defining the reduced regression model, we fitted it to the training data using the `fit` method and obtained predictions using the `predict` method. For the latter, we had to provide the forecasting horizon object as the argument. Alternatively, we could have provided a list/array of steps for which we wanted to obtain the forecasts.

In *Step 5*, we evaluated the forecast by calculating the MAPE score and plotting the forecasts compared to the actual values. To calculate the error metric, we used `sktime`'s `mean_absolute_percentage_error` function. An additional benefit of using `sktime`'s implementation is that we can easily calculate the **symmetric MAPE** (**sMAPE**) by specifying `symmetric=True` while calling the function.

At this point, we have noticed that the reduced regression model is suffering from the problem mentioned in the introduction—it does not capture the trend and seasonality of the time series. Hence, in the next steps, we showed how to deseasonalize and detrend the time series before using the reduced regression approach.

In *Step 6*, we deseasonalized the original time series. First, we instantiated the `Deseasonalizer` transformer. We indicated that there is monthly seasonality by providing `sp=12` and chose additive seasonality, as the magnitude of seasonal patterns does not seem to change over time. Under the hood, the `Deseasonalizer` class carries out the seasonal decomposition available in the `statsmodels` library (we covered it in the *Time series decomposition* recipe in the previous chapter) and removes the seasonal component from the time series. To fit the transformer and obtain the deseasonalized time series in a single step, we used the `fit_transform` method. After fitting the transformer, the seasonal component can be inspected by accessing the `seasonal_` attribute.

In *Step 7*, we removed the trend from the deseasonalized time series. First, we instantiated the `PolynomialTrendForecaster` class and specified `degree=1`. By doing so, we indicated that we were interested in a linear trend. Then, we passed the instantiated class to the `Detrender` transformer. Using the already familiar `fit_transform` method, we removed the trend from the deseasonalized time series.

In *Step 8*, we combined all the steps into a pipeline. We instantiated the `TransformedTargetForecaster` class, which is used when we first transform the time series and only then fit an ML model to create a forecast. As the `steps` argument, we provided a list of tuples, each of those containing the name of the step and the transformer/estimator used for carrying it out. In this pipeline, we chained deseasonalizing, detrending, and the reduced Random Forest model we have already used in *Step 4*. Then, we fitted the entire pipeline to the training data and obtained the predictions. In *Step 9*, we evaluated the pipeline's performance by calculating the MAPE and plotting the forecasts versus the actuals.



In this example, we only focused on creating the model using the original time series. Naturally, we can also have other features used for making predictions. `sktime` also offers functionalities to create pipelines containing relevant transformations for the regressors. Then, we should use the `ForecastingPipeline` class to apply the given transformers to `X` (features). We might also want to apply some transformations to `X` and other ones to the `y` (target). In such a case, we can pass the `TransformedTargetForecaster` containing any transformers that need to be applied to `y` as a step of the `ForecastingPipeline`.

In *Step 10*, we carried out an additional evaluation step. We used the walk-forward cross-validation using an expanding window to evaluate the model's performance. To define the cross-validation scheme, we used the `ExpandingWindowSplitter` class. As inputs, we had to provide:

- `fh`—The forecasting horizon. As we wanted to evaluate 12-steps-ahead forecasts, we provided a list of integers from 1 to 12.
- `initial_window`—The length of the initial training window. We set it to 60, which corresponds to 5 years of training data.
- `step_length`—This value indicates how many periods the expanding window is actually expanding by. We set it to 12, so each fold will have an extra year of training data.

After defining the validation scheme, we used the `evaluate` function to assess the performance of the pipeline defined in *Step 8*. While using the `evaluate` function, we also had to specify the `strategy` argument, which defined the approach to ingesting new data when the window expands. The options are as follows:

- `refit`—The model is refitted in each training window.
- `update`—The forecaster is updated with the new training in the window, but it is not refitted.
- `no-update_params`—The model is fitted to the first training window, and then it is reused without fitting or updating the model.

In *Step 11*, we used the `plot_series` function combined with a list comprehension to plot the original time series and the predictions obtained in each of the validation folds.

In the last two steps, we created and evaluated an ensemble model. First, we instantiated the `EnsembleForecaster` class and provided a list of tuples containing the names of the models and their respective classes/definitions. For this ensemble, we combined an AutoARIMA model with monthly seasonality (a SARIMA model) and the reduced Random Forest pipeline defined in *Step 8*. Additionally, we used the default value of the `aggfunc` argument, which is "`mean`". The argument determines the aggregation strategy used to create the final forecasts. In this case, the prediction of the ensemble model was the average of the predictions of the individual models. Other options include taking the median, minimum, or maximum values.

After instantiating the model, we used the already familiar `fit` and `predict` methods to fit the model and obtain the predictions.

There's more...

In this recipe, we covered reduced regression using `sktime`. As we have already mentioned, `sktime` is a framework offering all the tools you might need while working with time series. Below, we list some of the advantages of using `sktime` and its features:

- The library is suitable not only for working with time series forecasting but also regression, classification, and clustering. Additionally, it also provides feature extraction functionalities.
- `sktime` offers a few naive models, which are very useful for creating benchmarks. For example, we can use the `NaiveForecaster` model to create forecasts that are simply the last known value. Alternatively, we can use the last known seasonal value, for example, the forecast for January 2019 would be the value of the time series in January 2018.
- It provides a unified API as a wrapper around many popular time series libraries, such as `statsmodels`, `pmdarima`, `tbats`, or Meta's Prophet. To inspect all the available forecasting models, we can execute the `all_estimators("forecaster", as_dataframe=True)` command.
- By using reduction, it is possible to forecast using all the estimators compatible with the `scikit-learn` API.
- `sktime` provides functionalities for hyperparameter tuning with temporal cross-validation. Additionally, we can also tune hyperparameters connected to the reduction process, such as the number of lags or the window length.
- The library offers a wide range of performance evaluation metrics (not available in `scikit-learn`) and allows us to easily create custom scorers.
- The library extends `scikit-learn`'s pipelines to combine multiple transformers (detrending, deseasonalizing, and so on) with forecasting algorithms.

- The library provides AutoML capabilities to automatically determine the best forecaster from a wide range of models and their hyperparameters.

See also

- Löning, M., Bagnall, A., Ganesh, S., Kazakov, V., Lines, J., & Király, F. J. 2019. sktime: A Unified Interface for Machine Learning with Time Series. *arXiv preprint arXiv:1909.07872*.

Forecasting with Meta's Prophet

In the previous recipe, we showed how to reframe a time series forecasting problem in order to use popular machine learning models that are commonly used for regression tasks. This time, we present a model specifically designed for time series forecasting.

Prophet was introduced by Facebook (now Meta) back in 2017 and since then, it has become a very popular tool for time series forecasting. Some of the reasons for its popularity:

- Most of the time, it produces reasonable results/forecasts out of the box.
- It was designed to forecast business-related time series.
- It works best with daily time series with a strong seasonal component and at least a few seasons of training data.
- It can model any number of seasonalities (such as hourly, daily, weekly, monthly, quarterly, or yearly).
- The algorithm is quite robust to missing data and shifts in trend (it uses automatic changepoint detection for that).
- It easily accounts for holidays and special events.
- Compared to autoregressive models (such as ARIMA), it does not require stationary time series.
- We can employ business/domain knowledge to tune the forecasts by adjusting the human-interpretable hyperparameters of the model.
- We can use additional regressors to improve the model's predictive performance.



Naturally, the model is by no means perfect and it suffers from its own set of issues. In the *See also* section, we listed a few references showing the model's weaknesses.

The creators of Prophet approached the time series forecasting problem as a curve-fitting exercise (which raises quite a lot of controversies in the data science community) rather than explicitly looking at the time-based dependencies of each observation within a time series. As a result, Prophet is an additive model (a form of generalized additive models or GAMs) and can be presented as follows:

$$y(t) = g(t) + h(t) + s(t) + \varepsilon_t$$

where:

- $g(t)$ —Growth term, which is piecewise linear, logistic, or flat. The trend component models the non-periodic changes in the time series.
- $h(t)$ —Describes the effects of holidays and special days (which potentially occur on an irregular basis). They are added to the model as dummy variables.
- $s(t)$ —Describes various seasonal patterns modeled using the Fourier series.
- ε_t —Error term, which is assumed to be normally distributed.



The logistic growth trend is especially useful for modeling saturated (or capped) growth. For example, when we are forecasting the number of customers in a given country, we should not forecast more than the total number of the country's inhabitants. With Prophet, we can also account for the saturating minimum.

GAMs are simple yet powerful models that are gaining popularity. They assume that relationships between individual features and the target follow smooth patterns. Those can be linear or non-linear. Then, those relationships can be estimated simultaneously and added up to create the models' predicted values. For example, modeling seasonality as an additive component is the same approach as the one taken in Holt-Winters' exponential smoothing method. The GAM formulation used by Prophet has its advantages. First, it decomposes easily. Second, it accommodates new components, for example, when we identify a new source of seasonality.

Another important aspect of Prophet is the inclusion of changepoints in the process of estimating the trend, which makes the trend curve more flexible. Thanks to changepoints, the trend can be adjusted to sudden changes in the patterns, for example, the changes to sales patterns caused by the COVID pandemic. Prophet has an automatic procedure for detecting changepoints, but it can also accept manual inputs in the form of dates.

Prophet is estimated using a Bayesian approach (thanks to using Stan, which is a programming language for statistical inference written in C++), which allows for automatic changepoint selection, creating confidence intervals using methods like **Markov Chain Monte Carlo (MCMC)** or the **Maximum A Posteriori (MAP)** estimate.

In this recipe, we show how to forecast daily gold prices using data from the years 2015 to 2019. While we very well realize that the model will be unlikely to accurately forecast the gold prices, we use them as an illustration of how to train and use the model.

How to do it...

Execute the following steps to forecast daily gold prices with the Prophet model:

1. Import the libraries and authenticate with Nasdaq Data Link:

```
import pandas as pd
import nasdaqdatalink
from prophet import Prophet
from prophet.plot import add_changepoints_to_plot

nasdaqdatalink.ApiConfig.api_key = "YOUR_KEY_HERE"
```

2. Download the daily gold prices:

```
df = nasdaqdatalink.get(
    dataset="WGC/GOLD_DAILY_USD",
    start_date="2015-01-01",
    end_date="2019-12-31"
)

df.plot(title="Daily gold prices (2015-2019)")
```

Executing the snippet generates the following plot:

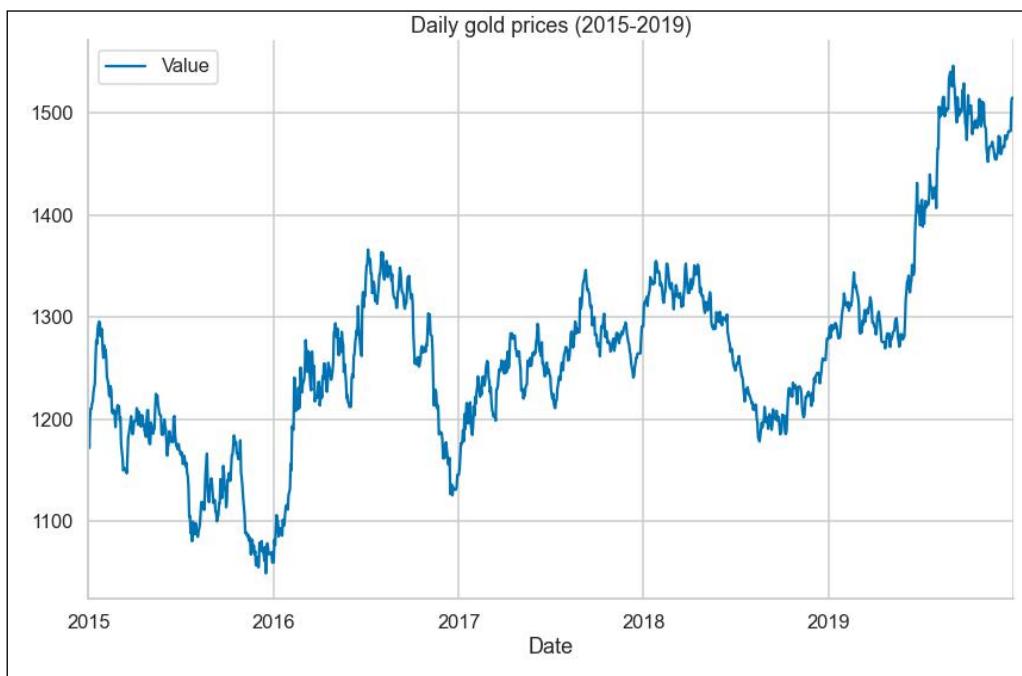


Figure 7.31: Daily gold prices from the years 2015 to 2019

3. Rename the columns:

```
df = df.reset_index(drop=False)
df.columns = ["ds", "y"]
```

4. Split the series into the training and test sets:

```
train_indices = df["ds"] < "2019-10-01"
df_train = df.loc[train_indices].dropna()
df_test = (
    df
    .loc[~train_indices]
    .reset_index(drop=True)
)
```

We arbitrarily chose to use the last quarter of 2019 as the test set. Hence, we will create a model forecasting around 60 observations in the future.

5. Create the instance of the model and fit it to the data:

```
prophet = Prophet(changepoint_range=0.9)
prophet.add_country_holidays(country_name="US")
prophet.add_seasonality(
    name="monthly", period=30.5, fourier_order=5
)
prophet.fit(df_train)
```

6. Forecast the gold prices for the fourth quarter of 2019 and plot the results:

```
df_future = prophet.make_future_dataframe(
    periods=len(df_test), freq="B"
)
df_pred = prophet.predict(df_future)
prophet.plot(df_pred)
```

Executing the snippet generates the following plot:

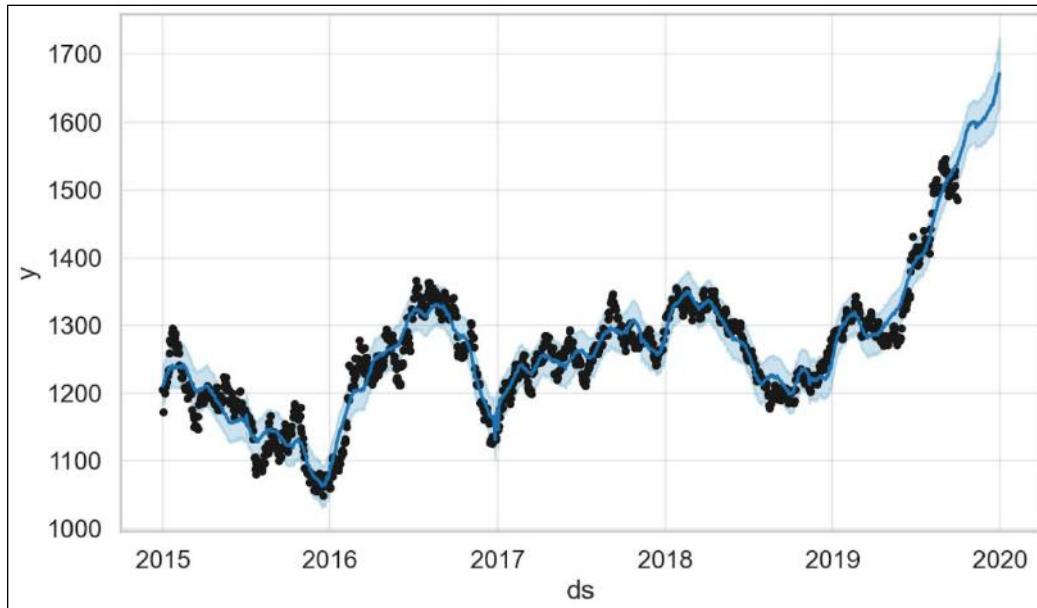


Figure 7.32: The forecast obtained using Prophet

To interpret the figure, we should know that:

- The black dots are the actual observations of the gold price.
- The blue line representing the fit does not match the observations exactly, as the model smooths out the noise in the data (also reducing the chance of overfitting).
- Prophet attempts to quantify uncertainty, which is represented by the light blue intervals around the fitted line. The interval is calculated assuming that the average frequency and magnitude of trend changes in the future will be the same as in the historical data.



It is also possible to create an interactive plot using `plotly`. To do so, we need to use the `plot_plotly` function instead of the `plot` method.

Additionally, it is worth mentioning that the prediction DataFrame contains quite a lot of columns with potentially useful information:

```
df_pred.columns
```

Using the snippet, we can see all the columns:

```
['ds', 'trend', 'yhat_lower', 'yhat_upper', 'trend_lower',
 'trend_upper', 'Christmas Day', 'Christmas Day_lower',
 'Christmas Day_upper', 'Christmas Day (Observed)',
 'Christmas Day (Observed)_lower', 'Christmas Day (Observed)_upper',
 'Columbus Day', 'Columbus Day_lower', 'Columbus Day_upper',
 'Independence Day', 'Independence Day_lower',
 'Independence Day_upper', 'Independence Day (Observed)',
 'Independence Day (Observed)_lower',
 'Independence Day (Observed)_upper', 'Labor Day', 'Labor Day_lower',
 'Labor Day_upper', 'Martin Luther King Jr. Day',
 'Martin Luther King Jr. Day_lower',
 'Martin Luther King Jr. Day_upper',
 'Memorial Day', 'Memorial Day_lower', 'Memorial Day_upper',
 'New Year's Day', 'New Year's Day_lower', 'New Year's Day_upper',
 'New Year's Day (Observed)', 'New Year's Day (Observed)_lower',
 'New Year's Day (Observed)_upper', 'Thanksgiving',
 'Thanksgiving_lower', 'Thanksgiving_upper', 'Veterans Day',
 'Veterans Day_lower', 'Veterans Day_upper',
 'Veterans Day (Observed)', 'Veterans Day (Observed)_lower',
 'Veterans Day (Observed)_upper', 'Washington's Birthday',
 'Washington's Birthday_lower', 'Washington's Birthday_upper',
 'additive_terms', 'additive_terms_lower', 'additive_terms_upper',
 'holidays', 'holidays_lower', 'holidays_upper', 'monthly',
 'monthly_lower', 'monthly_upper', 'weekly', 'weekly_lower',
 'weekly_upper', 'yearly', 'yearly_lower', 'yearly_upper',
 'multiplicative_terms', 'multiplicative_terms_lower',
 'multiplicative_terms_upper', 'yhat']
```

By analyzing the list, we can see all the components returned by the Prophet model. Naturally, we see the forecast (`yhat`) and its corresponding confidence intervals ('`yhat_lower`' and '`yhat_upper`'). Additionally, we see all the individual components of the model (such as trends, holiday effects, and seasonalities) together with their confidence intervals. Those might be interesting to us because of the following considerations:

- As Prophet is an additive model, we can sum up all the components to arrive at the final forecast. Hence, we can look at those values as a type of feature importance, which can be used to explain the forecast.
- We could also use the Prophet model to obtain those component values and then feed them to another model (for example, a tree-based model) as features.

7. Add changepoints to the plot:

```
fig = prophet.plot(df_pred)
a = add_changepoints_to_plot(
    fig.gca(), prophet, df_pred
)
```

Executing the snippet generates the following plot:

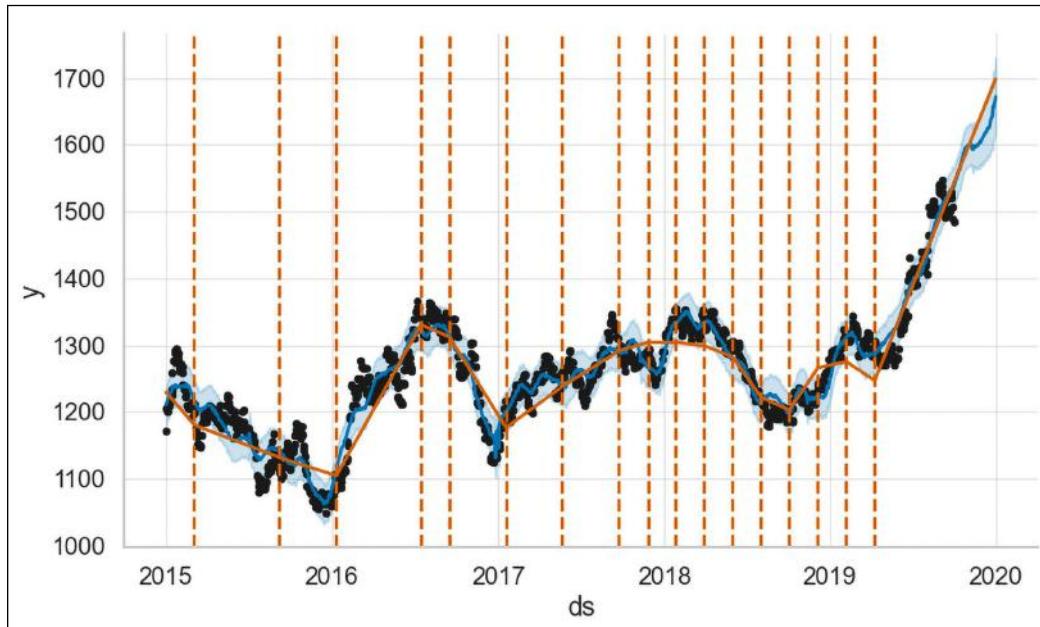


Figure 7.33: The model's fit together with the identified changepoints

We can also look up the exact dates that were identified as changepoints using the `changepoints` method of a fitted Prophet model.

8. Inspect the decomposition of the time series:

```
prophet.plot_components(df_pred)
```

Executing the snippet generates the following plot:

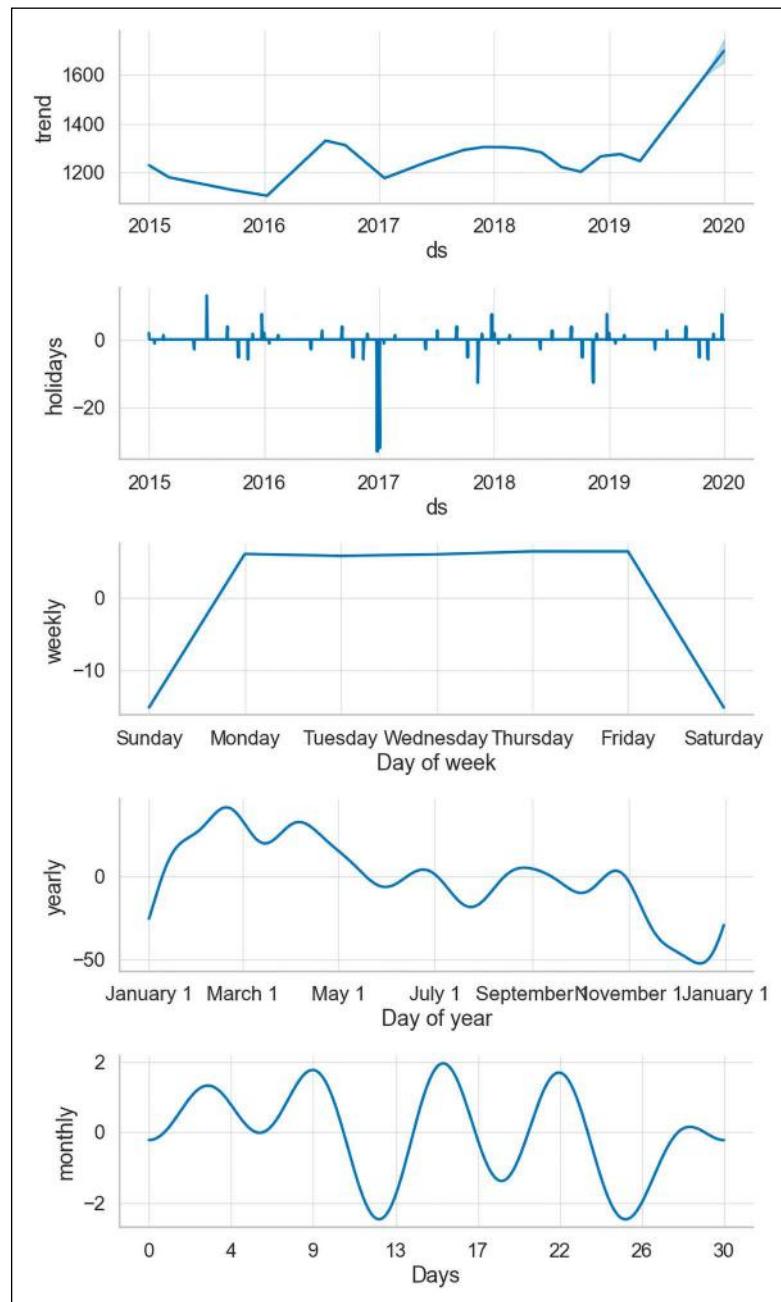


Figure 7.34: The decomposition plot showing the individual components of the Prophet model

We do not spend much time inspecting the components, as the time series of gold prices probably does not have many seasonal effects or should not be impacted by the US holidays. That is especially true for the holidays, as the stock market is closed on major holidays. Therefore, the effect of these holidays may be reflected by the market on the days before and after. As we have mentioned before, we are aware of that and we just wanted to show how Prophet works.

One thing to note is that the weekly seasonality is noticeably different for Saturday and Sunday. That is caused by the fact that the gold prices are collected during weekdays. Hence, we can safely ignore the weekend patterns.

However, it is interesting to observe the trend component, which we can also see plotted in *Figure 7.33*, together with the detected changepoints.

9. Merge the test set with the forecasts:

```
SELECTED_COLS = [
    "ds", "yhat", "yhat_lower", "yhat_upper"
]

df_pred = (
    df_pred
    .loc[:, SELECTED_COLS]
    .reset_index(drop=True)
)
df_test = df_test.merge(df_pred, on=["ds"], how="left")
df_test["ds"] = pd.to_datetime(df_test["ds"])
df_test = df_test.set_index("ds")
```

10. Plot the test values vs. predictions:

```
fig, ax = plt.subplots(1, 1)

PLOT_COLS = [
    "y", "yhat", "yhat_lower", "yhat_upper"
]
ax = sns.lineplot(data=df_test[PLOT_COLS])
ax.fill_between(
    df_test.index,
    df_test["yhat_lower"],
    df_test["yhat_upper"],
    alpha=0.3
)
```

```

ax.set(
    title="Gold Price - actual vs. predicted",
    xlabel="Date",
    ylabel="Gold Price ($)"
)

```

Executing the snippet generates the following plot:

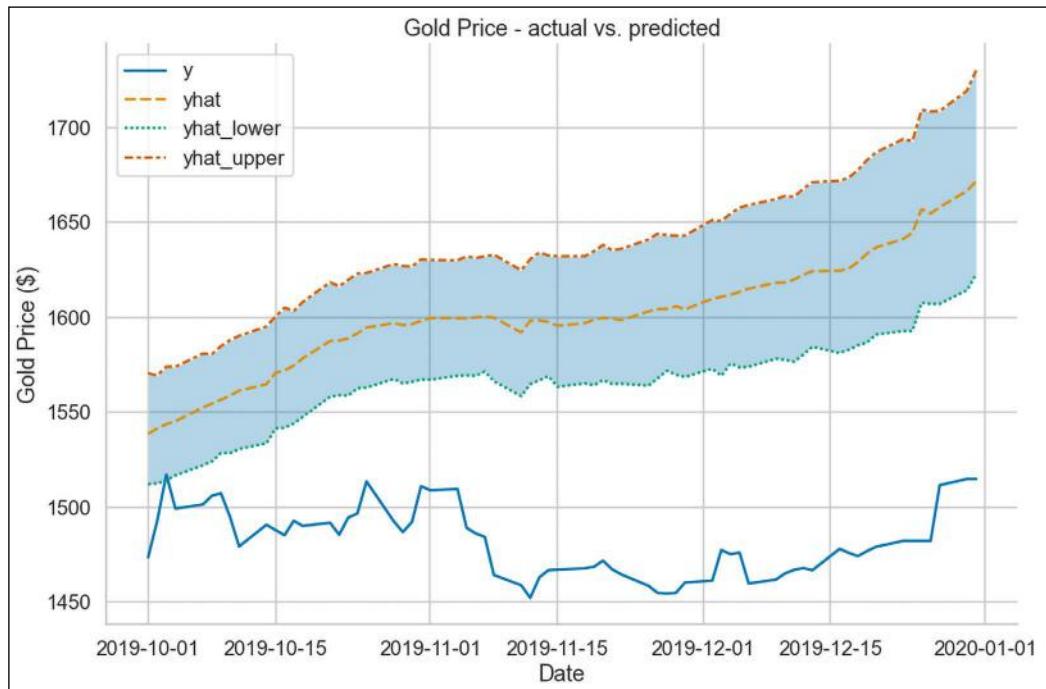


Figure 7.35: Forecast vs ground truth

As we can see in *Figure 7.35*, the model's prediction is quite off. As a matter of fact, the 80% confidence interval (the default setting, we can change it using the `interval_width` hyperparameter) does not capture almost any of the actual values.

How it works...

After importing the libraries, we downloaded the daily gold prices from Nasdaq Data Link.

In *Step 3*, we renamed the columns of the DataFrame in order to make it compatible with Prophet. The algorithm requires two columns:

- `ds`—Indicating the timestamp
- `y`—The target variable

In *Step 4*, we split the DataFrame into training and test sets. We arbitrarily chose to use the fourth quarter of 2019 as the test set.

In *Step 5*, we instantiated the Prophet model. While doing so, we specified a few settings:

- We set `changepoint_range` to `0.9`, which means that the algorithm can identify changepoints in the first 90% of the training dataset. By default, Prophet adds 25 changepoints in the first 80% of the time series. In this case, we wanted to capture the more recent trends as well.
- We added the monthly seasonality by using the `add_seasonality` method with values suggested by Prophet's documentation. Specifying `period` as `30.5` means that we expect the patterns to repeat themselves after roughly 30.5 days. The other parameter—`fourier_order`—can be used to specify the number of Fourier terms that are used to build the particular seasonal component (in this case, monthly). In general, the higher the order, the more flexible the seasonality component.
- We used the `add_country_holidays` method to add the US holidays to the model. We have used the default calendar (available via the `holidays` library), but it is also possible to add custom events that are not available in the calendar. One example might be Black Friday. It is also worth mentioning that when providing the custom events, we can also specify if we expect the surrounding days to be affected as well. For example, in a retail scenario, we might expect the traffic/sales to be lower in the days following Christmas. On the other hand, we might expect a peak just before Christmas.

Then, we fitted the model using the `fit` method.

In *Step 6*, we used the fitted model to obtain predictions. To create forecasts with Prophet, we had to create a special DataFrame using the `make_future_dataframe` method. While doing so, we indicated that we want to forecast for the length of the test set (by default, this is measured in days) and that we wanted to use business days. That part is important, as we do not have gold prices for the weekends. Then, we created the predictions using the `predict` method of the fitted model.

In *Step 7*, we added the identified changepoints to the plot using the `add_changepoints_to_plot` function. One thing to note here is that we had to use the `gca` method of the created figure to get its current axis. We had to use it to correctly identify to which plot we wanted to add the changepoints.

In *Step 8*, we inspected the components of the model. To do so, we used the `plot_components` method with the prediction DataFrame as the method's argument.

In *Step 9*, we merged the test set with the prediction DataFrame. We used a left join, which returns all the rows from the left table (test set) and the matched rows from the right table (prediction DataFrame) while leaving the unmatched rows empty.

Finally, we plotted the predictions (together with the confidence intervals) and the ground truth to visually evaluate the model's performance.

There's more...

Prophet offers quite a lot of interesting functionalities. While it is definitely too much to mention in a single recipe, we wanted to highlight two things.

Built-in cross-validation

In order to properly evaluate the model's performance (and potentially tune its hyperparameters), we do need a validation framework. Prophet implements the already familiar walk-forward cross-validation in its `cross_validation` function. In this subsection, we show how to use it:

1. Import the libraries:

```
from prophet.diagnostics import (cross_validation,
                                  performance_metrics)
from prophet.plot import plot_cross_validation_metric
```

2. Run Prophet's cross-validation:

```
df_cv = cross_validation(
    prophet,
    initial="756 days",
    period="60 days",
    horizon = "60 days"
)

df_cv
```

We have specified that we want:

- The initial window to contain 3 years of data (a year contains approximately 252 trading days)
- A forecast horizon of 60 days
- The forecasts to be calculated every 60 days

Executing the snippet generates the following output:

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	2017-02-13	1240.428658	1214.075002	1266.887144	1222.25	2017-02-12
1	2017-02-14	1244.035274	1218.709804	1269.177963	1230.75	2017-02-12
2	2017-02-15	1245.784079	1219.778945	1270.415037	1224.40	2017-02-12
3	2017-02-16	1247.101206	1222.243635	1272.417761	1240.55	2017-02-12
4	2017-02-17	1247.049617	1219.427610	1273.846214	1241.95	2017-02-12
...
681	2019-09-24	1411.521433	1376.789748	1445.763547	1520.65	2019-08-01
682	2019-09-25	1411.647219	1378.678598	1447.529225	1528.75	2019-08-01
683	2019-09-26	1411.175984	1377.546582	1449.880765	1506.40	2019-08-01
684	2019-09-27	1409.462673	1374.309112	1445.839082	1489.90	2019-08-01
685	2019-09-30	1408.982620	1372.462776	1448.030593	1485.30	2019-08-01

Figure 7.36: The output of Prophet's cross-validation

The DataFrame contains the predictions (including the confidence intervals) and the actual value for a combination of cutoff dates (the last time point in the training set used to generate the forecast) and ds dates (the date in the validation set for which the forecast was generated). In other words, the procedure creates a forecast for every observed point between cutoff and cutoff + horizon.

The algorithm also informed us what it was going to do:

```
Making 16 forecasts with cutoffs between 2017-02-12 00:00:00 and 2019-08-01 00:00:00
```

3. Calculate the aggregated performance metrics:

```
df_p = performance_metrics(df_cv)
df_p
```

Executing the snippet generates the following output:

	horizon	mse	rmse	mae	mape	mdape	smape	coverage
0	6 days	1748.610554	41.816391	31.559107	0.024066	0.018103	0.024227	0.551471
1	7 days	1989.890473	44.608188	33.977247	0.025884	0.020106	0.026108	0.529412
2	8 days	2202.222234	46.927841	35.882566	0.027195	0.022757	0.027480	0.491979
3	9 days	2367.956551	48.661654	38.572966	0.029299	0.024370	0.029526	0.459559
4	10 days	2529.118619	50.290343	41.208362	0.031576	0.031014	0.031808	0.397059
5	11 days	2691.744910	51.882029	42.421694	0.032552	0.031014	0.032844	0.367647
6	12 days	2725.613313	52.207407	43.137264	0.033074	0.030830	0.033348	0.362299
7	13 days	2937.700093	54.200554	44.898209	0.034387	0.032834	0.034652	0.333333
8	14 days	3270.445886	57.187812	47.956755	0.036807	0.034913	0.037067	0.294118
9	15 days	3656.224802	60.466725	50.168416	0.038266	0.035763	0.038576	0.288770

Figure 7.37: The first 10 lines of the performance overview

Figure 7.37 presents the first 10 rows of the DataFrame containing the aggregated performance scores from our cross-validation. As per our cross-validation scheme, the entire DataFrame contains all the horizons until 60 days.



Please refer to Prophet's documentation for the exact logic behind the aggregated performance metrics generated by the `performance_metrics` function.

4. Plot the MAPE score:

```
plot_cross_validation_metric(df_cv, metric="mape")
```

Executing the snippet generates the following plot:

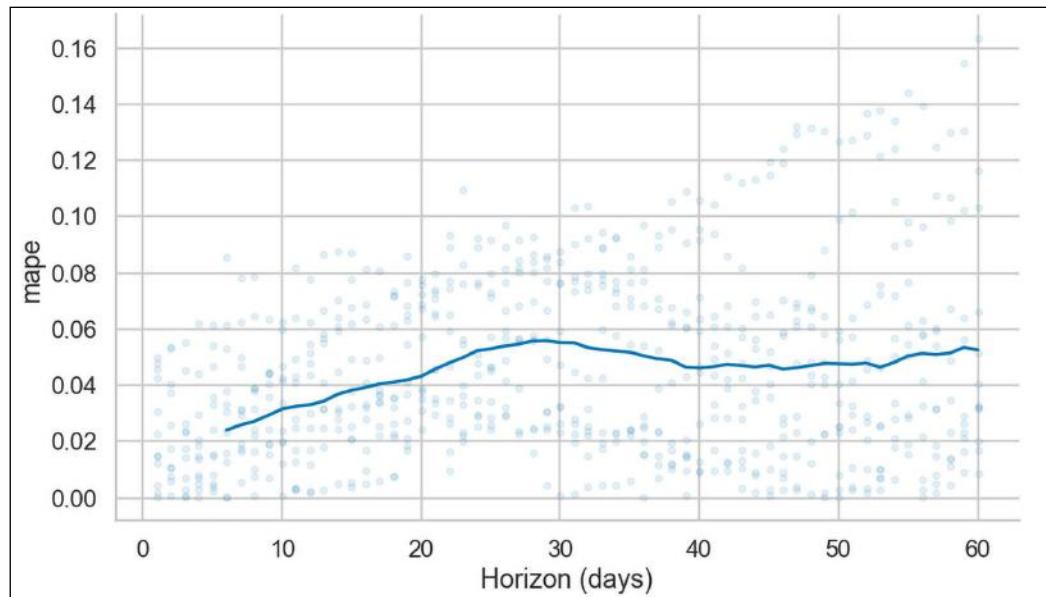


Figure 7.38: The MAPE score over horizons

The dots in *Figure 7.38* represent the absolute percent error for each prediction in the cross-validation DataFrame. The blue line represents the MAPE. The average is taken over a rolling window of the dots. For more information about the rolling window, please refer to Prophet's documentation.

Tuning the model

As we have already seen, Prophet has quite a few tunable hyperparameters. The authors of the library suggest that the following hyperparameters might be worth tuning in order to achieve a better fit:

- `changepoint_prior_scale`—Possibly the most impactful hyperparameter, which determines the flexibility of the trend. Particularly, how much the trend changes at the trend changepoints. A too-small value will make the trend less flexible and might cause the trend to underfit, while a too-large value might cause the trend to overfit (and potentially capture the yearly seasonality as well).
- `seasonality_prior_scale`—A hyperparameter controlling the flexibility of the seasonality terms. Large values allow the seasonality to fit significant fluctuations, while small values shrink the seasonality's magnitude. The default value of 10 applies basically no regularization.
- `holidays_prior_scale`—Very similar to `seasonality_prior_scale`, but controls the flexibility to fit holiday effects.
- `seasonality_mode`—We can choose either additive or multiplicative seasonality. The best way to choose this one is to inspect the time series and see if the magnitude of seasonal fluctuations grows with the passage of time.

- `changepoint_range`—This parameter corresponds to the percentage of the time series in which the algorithm can identify changepoints. A rule of thumb to identify a good value for this hyperparameter is to look at the model's fit in the last `1-changepoint_range` percent of training data. If the model is doing a bad job there, we might want to increase the value of the hyperparameter.

As in the other cases, we might want to use a procedure like grid search (combined with cross-validation) to identify the best set of hyperparameters, while trying to avoid/minimize the risk of overfitting to the training data.

See also

- Rafferty, G. 2021. *Forecasting Time Series Data with Facebook Prophet*. Packt Publishing Ltd.
- Taylor, S. J., & Letham, B. 2018. “Forecasting at scale,” *The American Statistician*, 72(1): 37-45.

AutoML for time series forecasting with PyCaret

We have already spent some time explaining how to build ML models for time series forecasting, how to create relevant features, and how to use dedicated models (such as Meta's Prophet) for the task. It is only fitting to conclude the chapter with an extension of all of the mentioned parts—an AutoML tool.

One of the available tools is PyCaret, which is an open-source, low-code ML library. The goal of the tool is to automate machine learning workflows. Using PyCaret, we can train and tune dozens of popular ML models with only a few lines of code. While it was originally built for classic regression and classification tasks, it also has a dedicated time series module, which we will present in this recipe.

The PyCaret library is essentially a wrapper around several popular machine learning libraries and frameworks such as `scikit-learn`, XGBoost, LightGBM, CatBoost, Optuna, Hyperopt, and a few more. And to be more precise, PyCaret's time series module is built on top of the functionalities provided by `sktime`, for example, its reduction framework and pipelining capabilities.

In this recipe, we will use the PyCaret library to find the best model for predicting the monthly US unemployment rate.

Getting ready

In this recipe, we will use the same dataset we have already used in the previous recipes. You can find more information on how to download and prepare the time series in the *Validation methods for time series* recipe.

How to do it...

Execute the following steps to forecast the US unemployment rates using PyCaret:

1. Import the libraries:

```
from pycaret.datasets import get_data
from pycaret.time_series import TSForecastingExperiment
```

2. Set up the experiment:

```
exp = TSForecastingExperiment()  
exp.setup(df, fh=6, fold=5, session_id=42)
```

Executing the snippet generates the following experiment summary:

	Description	Value
0	session_id	42
1	Target	unemp_rate
2	Approach	Univariate
3	Exogenous Variables	Not Present
4	Original data shape	(120, 1)
5	Transformed data shape	(120, 1)
6	Transformed train set shape	(114, 1)
7	Transformed test set shape	(6, 1)
8	Rows with missing values	0.0%
9	Fold Generator	ExpandingWindowSplitter
10	Fold Number	5
11	Enforce Prediction Interval	0
12	Seasonal Period(s) Tested	12
13	Seasonality Present	1
14	Seasonalities Detected	[12]
15	Primary Seasonality	12
16	Target Strictly Positive	True
17	Target White Noise	No
18	Recommended d	1
19	Recommended Seasonal D	0
20	Preprocess	0
21	CPU Jobs	-1
22	Use GPU	0
23	Log Experiment	0
24	Experiment Name	ts-default-name
25	USI	2a00

Figure 7.39: The summary of PyCaret's experiment

We can see that the library automatically took the last 6 observations as the test set and identified monthly seasonality in the provided time series.

3. Explore the time series using visualizations:

```
exp.plot_model(
    plot="diagnostics",
    fig_kwarg={"height": 800, "width": 1000}
)
```

Executing the snippet generates the following plot:

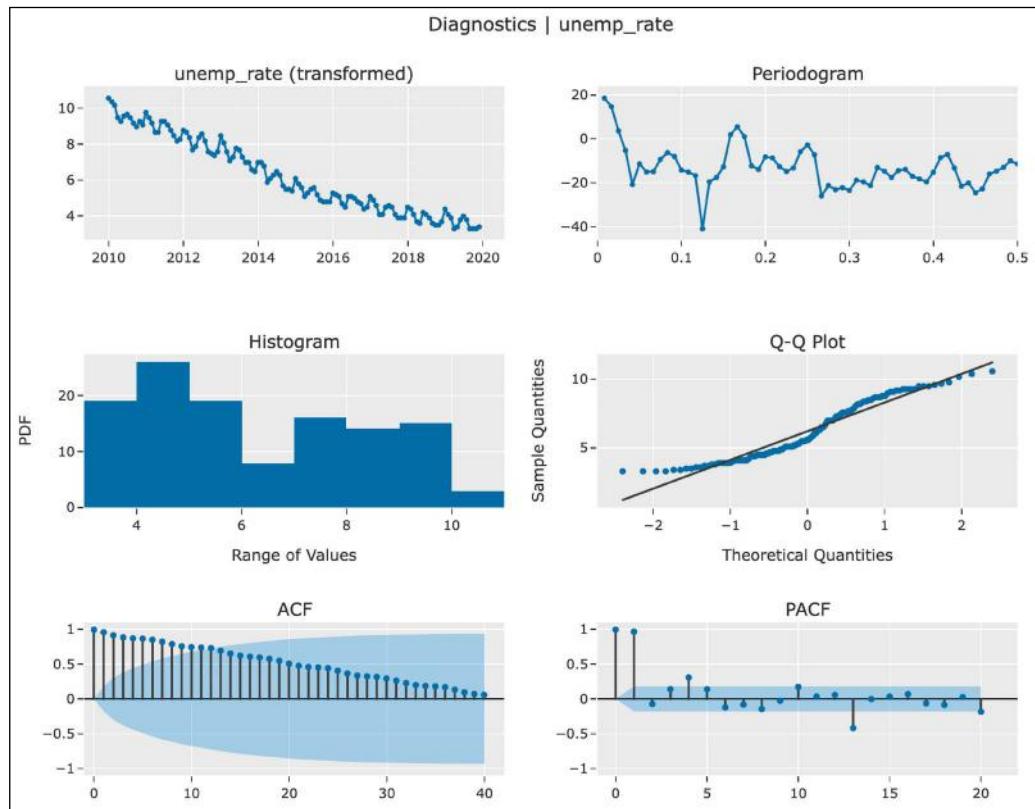


Figure 7.40: Diagnostics plots of the time series

While most of the plots are already familiar, the new one is the periodogram. We can use it (together with the fast Fourier transform plot) to study the frequency components of the analyzed time series. While this might be outside of the scope of this book, we can mention the following highlights of interpreting those plots:

- Peaking around 0 can indicate the need to difference the time series. It could be indicative of a stationary ARMA process.
- Peaking at some frequency and its multiples indicates seasonality. The lowest of those frequencies is called the **fundamental frequency**. Its inverse is the seasonal period of the model. For example, a fundamental frequency of 0.0833 corresponds to the seasonal period of 12, as $1/0.0833 = 12$.

Using the following snippet, we can visualize the cross-validation scheme that will be used for the experiment:

```
exp.plot_model(plot="cv")
```

Executing the snippet generates the following plot:

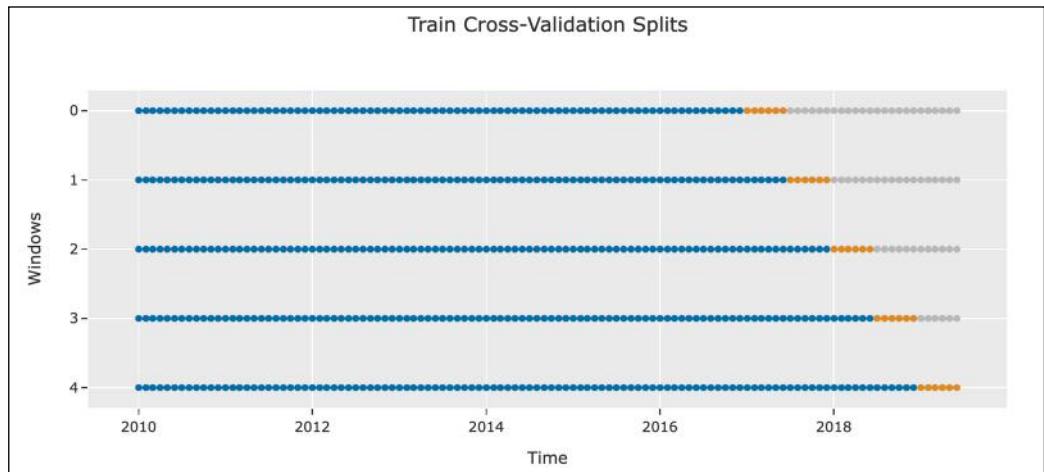


Figure 7.41: An example of 5-fold walking cross-validation using expanding window



In the accompanying notebook, we also show some of the other available plots, for example, seasonal decomposition, **fast Fourier transform (FFT)**, and more.

4. Run statistical tests on the time series:

```
exp.check_stats()
```

Executing the snippet generates the following DataFrame with the results of various tests:

	Test	Test Name	Data	Property	Setting	Value
0	Summary	Statistics	Transformed	Length		120.0
1	Summary	Statistics	Transformed	# Missing Values		0.0
2	Summary	Statistics	Transformed	Mean		6.225
3	Summary	Statistics	Transformed	Median		5.6
4	Summary	Statistics	Transformed	Standard Deviation		2.10223
5	Summary	Statistics	Transformed	Variance		4.41937
6	Summary	Statistics	Transformed	Kurtosis		-1.232708
7	Summary	Statistics	Transformed	Skewness		0.345242
8	Summary	Statistics	Transformed	# Distinct Values		59.0
9	White Noise	Ljung-Box	Transformed	Test Statistic	{'alpha': 0.05, 'K': 24}	1610.583379
10	White Noise	Ljung-Box	Transformed	Test Statistic	{'alpha': 0.05, 'K': 48}	1778.242684
11	White Noise	Ljung-Box	Transformed	p-value	{'alpha': 0.05, 'K': 24}	0.0
12	White Noise	Ljung-Box	Transformed	p-value	{'alpha': 0.05, 'K': 48}	0.0
13	White Noise	Ljung-Box	Transformed	White Noise	{'alpha': 0.05, 'K': 24}	False
14	White Noise	Ljung-Box	Transformed	White Noise	{'alpha': 0.05, 'K': 48}	False
15	Stationarity	ADF	Transformed	Stationarity	{'alpha': 0.05}	False
16	Stationarity	ADF	Transformed	p-value	{'alpha': 0.05}	0.263656
17	Stationarity	ADF	Transformed	Test Statistic	{'alpha': 0.05}	-2.053411
18	Stationarity	ADF	Transformed	Critical Value 1%	{'alpha': 0.05}	-3.492996
19	Stationarity	ADF	Transformed	Critical Value 5%	{'alpha': 0.05}	-2.888955
20	Stationarity	ADF	Transformed	Critical Value 10%	{'alpha': 0.05}	-2.581393
21	Stationarity	KPSS	Transformed	Trend Stationarity	{'alpha': 0.05}	False
22	Stationarity	KPSS	Transformed	p-value	{'alpha': 0.05}	0.01
23	Stationarity	KPSS	Transformed	Test Statistic	{'alpha': 0.05}	0.44361
24	Stationarity	KPSS	Transformed	Critical Value 10%	{'alpha': 0.05}	0.119
25	Stationarity	KPSS	Transformed	Critical Value 5%	{'alpha': 0.05}	0.146
26	Stationarity	KPSS	Transformed	Critical Value 2.5%	{'alpha': 0.05}	0.176
27	Stationarity	KPSS	Transformed	Critical Value 1%	{'alpha': 0.05}	0.216
28	Normality	Shapiro	Transformed	Normality	{'alpha': 0.05}	False
29	Normality	Shapiro	Transformed	p-value	{'alpha': 0.05}	0.000005

Figure 7.42: DataFrame with the results of various statistical tests

We can also carry out only subsets of all the tests. For example, we can execute the summary tests using the following snippet:

```
exp.check_stats(test="summary")
```

5. Find the five best-fitting pipelines:

```
best_PIPELINES = exp.compare_models(
    sort="MAPE", turbo=False, n_select=5
)
```

Executing the snippet generates the following DataFrame with the performance overview:

Model	MASE	RMSSE	MAE	RMSE	MAPE	SMAPE	R2	TT (Sec)
bats	BATS	0.1222	0.1364	0.0911	0.1105	0.0225	0.0227	0.8702
tbats	TBATS	0.1308	0.1410	0.0976	0.1142	0.0239	0.0242	0.8641
auto_arima	Auto ARIMA	0.1487	0.1626	0.1110	0.1319	0.0280	0.0287	0.7229
prophet	Prophet	0.1599	0.1675	0.1201	0.1364	0.0293	0.0294	0.8053
theta	Theta Forecaster	0.1802	0.1906	0.1348	0.1547	0.0332	0.0332	0.7221
ets	ETS	0.1976	0.2108	0.1474	0.1708	0.0358	0.0364	0.6639
exp_smooth	Exponential Smoothing	0.2074	0.2256	0.1547	0.1828	0.0371	0.0381	0.6565
xgboost_cds_dt	Extreme Gradient Boosting w/ Cond. Deseasonalize & Detrending	0.2118	0.2510	0.1608	0.2056	0.0390	0.0403	0.4461
arima	ARIMA	0.2453	0.2573	0.1833	0.2088	0.0462	0.0477	0.4979
lr_cds_dt	Linear w/ Cond. Deseasonalize & Detrending	0.2516	0.2635	0.1907	0.2163	0.0465	0.0481	0.4679
huber_cds_dt	Huber w/ Cond. Deseasonalize & Detrending	0.2524	0.2662	0.1914	0.2185	0.0466	0.0482	0.4544
et_cds_dt	Extra Trees w/ Cond. Deseasonalize & Detrending	0.2726	0.2918	0.2087	0.2411	0.0497	0.0519	0.2802
br_cds_dt	Bayesian Ridge w/ Cond. Deseasonalize & Detrending	0.2715	0.2828	0.2056	0.2319	0.0501	0.0519	0.3964
ridge_cds_dt	Ridge w/ Cond. Deseasonalize & Detrending	0.2915	0.3005	0.2202	0.2460	0.0540	0.0560	0.3315
rf_cds_dt	Random Forest w/ Cond. Deseasonalize & Detrending	0.3052	0.3161	0.2322	0.2603	0.0558	0.0582	0.2598
ada_cds_dt	AdaBoost w/ Cond. Deseasonalize & Detrending	0.3063	0.3214	0.2333	0.2647	0.0566	0.0590	0.1843
dt_cds_dt	Decision Tree w/ Cond. Deseasonalize & Detrending	0.3072	0.3163	0.2326	0.2593	0.0567	0.0584	0.2933
lar_cds_dt	Least Angular Regressor w/ Cond. Deseasonalize & Detrending	0.3288	0.3381	0.2503	0.2786	0.0607	0.0634	0.0918
knn_cds_dt	K Neighbors w/ Cond. Deseasonalize & Detrending	0.3390	0.3410	0.2562	0.2792	0.0630	0.0656	0.1838
gbr_cds_dt	Gradient Boosting w/ Cond. Deseasonalize & Detrending	0.3448	0.3642	0.2596	0.2972	0.0640	0.0670	0.0070
omp_cds_dt	Orthogonal Matching Pursuit w/ Cond. Deseasonalize & Detrending	0.3563	0.3756	0.2700	0.3082	0.0657	0.0688	-0.960
lightgbm_cds_dt	Light Gradient Boosting w/ Cond. Deseasonalize & Detrending	0.3603	0.3727	0.2692	0.3028	0.0667	0.0698	0.0176
catboost_cds_dt	CatBoost Regressor w/ Cond. Deseasonalize & Detrending	0.3825	0.3896	0.2912	0.3208	0.0703	0.0740	-0.2316
naive	Naive Forecaster	0.5032	0.5268	0.3767	0.4284	0.0949	0.0926	-1.3404
snaive	Seasonal Naive Forecaster	0.5834	0.5586	0.4400	0.4570	0.1094	0.1030	-1.5401
croston	Croston	0.6327	0.6794	0.4762	0.5550	0.1232	0.1136	-2.7134
en_cds_dt	Elastic Net w/ Cond. Deseasonalize & Detrending	0.7809	0.7544	0.5822	0.6118	0.1451	0.1586	-3.1322
lasso_cds_dt	Lasso w/ Cond. Deseasonalize & Detrending	0.7809	0.7544	0.5822	0.6118	0.1451	0.1586	-3.1322
llar_cds_dt	Lasso Least Angular Regressor w/ Cond. Deseasonalize & Detrending	0.7809	0.7544	0.5822	0.6118	0.1451	0.1586	-3.1322
polytrend	Polynomial Trend Forecaster	0.8269	0.8207	0.6164	0.6650	0.1499	0.1651	-3.5825
par_cds_dt	Passive Aggressive w/ Cond. Deseasonalize & Detrending	1.3195	1.3532	0.9712	1.0288	0.2564	0.2742	-25.5411
grand_means	Grand Means Forecaster	3.7084	3.4400	2.7813	2.8005	0.6998	0.5138	-89.1440

Figure 7.43: DataFrame with the cross-validation scores of all the fitted models

Inspecting the `best_PIPELINES` object prints the best pipelines:

```
[BATS(show_warnings=False, sp=12, use_box_cox=True),
 TBATS(show_warnings=False, sp=[12], use_box_cox=True),
 AutoARIMA(random_state=42, sp=12, suppress_warnings=True),
 ProphetPeriodPatched(),
 ThetaForecaster(sp=12)]
```

6. Tune the best pipelines:

```
best_PIPELINES_tuned = [
    exp.tune_model(model) for model in best_PIPELINES
]
best_PIPELINES_tuned
```

After tuning, the best-performing pipelines are the following:

```
[BATS(show_warnings=False, sp=12, use_box_cox=True),
 TBATS(show_warnings=False, sp=[12], use_box_cox=True,
       use_damped_trend=True, use_trend=True),
 AutoARIMA(random_state=42, sp=12, suppress_warnings=True),
 ProphetPeriodPatched(changepoint_prior_scale=0.016439324494196616,
                       holidays_prior_scale=0.01095960453692584,
                       seasonality_prior_scale=7.886714129990491),
 ThetaForecaster(sp=12)]
```

Calling the `tune_model` method also prints out the cross-validation performance summary of each of the tuned models. For brevity, we do not print it here. However, you can inspect the accompanying notebook to see how the performance has changed as a result of tuning.

7. Blend the five tuned pipelines:

```
blended_model = exp.blend_models(
    best_PIPELINES_tuned, method="mean"
)
```

8. Create the predictions using the blended model and plot the forecasts:

```
y_pred = exp.predict_model(blended_model)
```

Executing the snippet also generates the test set performance summary:

	Model	MASE	RMSSE	MAE	RMSE	MAPE	SMAPE	R2
0	EnsembleForecaster	0.1707	0.1685	0.1174	0.1281	0.0328	0.0333	0.7898

Figure 7.44: The scores calculated using the predictions for the test set

Then, we plot the forecast for the test set:

```
exp.plot_model(estimator=blended_model)
```

Executing the snippet generates the following plot:

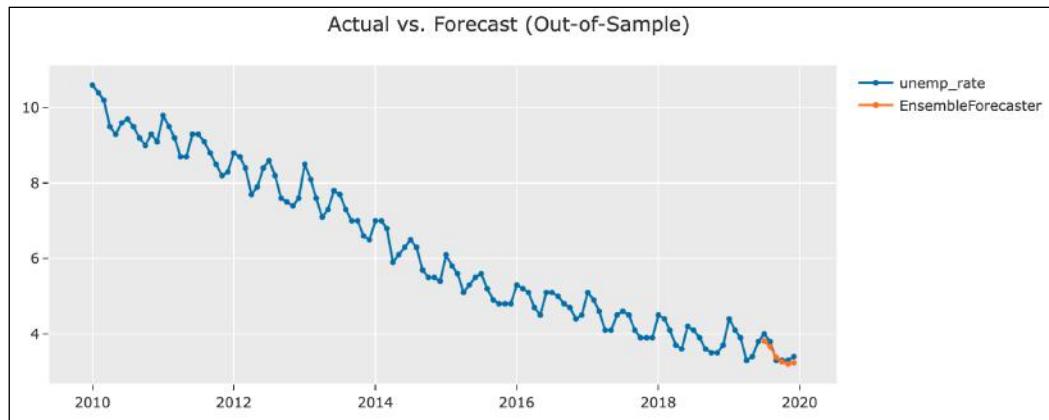


Figure 7.45: The time series, together with the predictions made for the test set

9. Finalize the model:

```
final_model = exp.finalize_model(blended_model)  
exp.plot_model(final_model)
```

Executing the snippet generates the following plot:

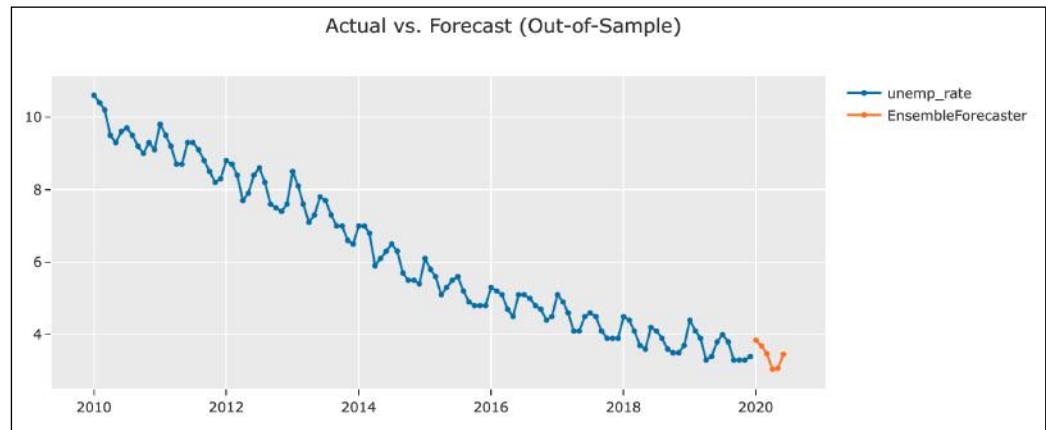


Figure 7.46: Out-of-sample prediction for the first 6 months of 2020

Just by looking at the plot, it seems that the forecasts are plausible and contain a clearly identifiable seasonal pattern. We can also generate and print the predictions we have already seen in the plot:

```
y_pred = exp.predict_model(final_model)  
print(y_pred)
```

Executing the snippet generates the following predictions for the next 6 months:

```
y_pred
2020-01  3.8437
2020-02  3.6852
2020-03  3.4731
2020-04  3.0444
2020-05  3.0711
2020-06  3.4585
```

How it works...

After importing the libraries, we set up the experiment. First, we instantiated an object of the `TSForecastingExperiment` class. Then, we used the `setup` method to provide the `DataFrame` with the time series, the forecast horizon, the number of cross-validation folds, and a session ID. For our experiment, we specified that we are interested in forecasting 6 months ahead and that we want to use 5 walk-forward cross-validation folds using an expanding window (the default variant). It is also possible to use the sliding window.



PyCaret offers two APIs: the functional one and the object-oriented one (using classes). In this recipe, we are presenting the latter.

While setting up the experiment, we could also indicate whether we want to apply some transformation to the target time series. We could select one of the following options: "box-cox", "log", "sqrt", "exp", "cos".



To extract the training and test sets from the experiment, we can use the following commands: `exp.get_config("y_train")` and `exp.get_config("y_test")`.

In *Step 3*, we carried out a quick EDA of the time series using the `plot_model` method of the `TSForecastingExperiment` object. To generate different plots, we simply changed the `plot` argument of the method.

In *Step 4*, we looked into a variety of statistical tests using the `check_stats` method of the `TSForecastingExperiment` class.

In *Step 5*, we used the `compare_models` method to train a selection of statistical and machine learning models and evaluate their performance using the selected cross-validation scheme. We indicated that we wanted to select the five best pipelines based on the MAPE score. We set `turbo=False` to also train models that might be a bit more time-consuming to train (for example, Prophet, BATS, and TBATS).



PyCaret uses the concept of pipelines as sometimes the “model” is actually built from several steps. For example, we might first detrend and deseasonalize the time series before fitting a regression model. For example, a `Random Forest w/ Cond. Deseasonalize & Detrending` model is a `sktime` pipeline that first conditionally deseasonalizes the time series. Afterward, detrending is applied, and only then, the reduced Random Forest is fitted. The conditional part by deseasonalizing refers to first checking with a statistical test if there is seasonality in the time series. If it is detected, then deseasonalization is applied.

There are a few things worth mentioning at this step:

- We can extract the `DataFrame` with the performance comparison using the `pull` method.
- We can use the `models` method to print a list of all the available models, together with their reference (to the original library, as PyCaret is a wrapper), and an indication of whether the model is taking more time to train and is hidden behind the `turbo` flag.
- We can also decide whether we only want to train some of the models (using the `include` argument of the `compare_models` method) or whether we want to train all the models except for a selected few (using the `exclude` argument).

In *Step 6*, we tuned the best pipelines. To do so, we used list comprehension to iterate over the identified pipelines and then used the `tune_model` method to carry out hyperparameter tuning. By default, it uses a randomized grid search (more on that in *Chapter 13, Applied Machine Learning: Identifying Credit Default*) using a grid of hyperparameters provided by the authors of the library. Those work as a good starting point, and in case we want to adjust them, we can easily do so.

In *Step 7*, we create an ensemble model, which is a blend of the five best pipelines (tuned versions). We decided to take the mean of the forecasts created by the individual models. Alternatively, we could use the median or voting. The latter is a voting scheme in which each model is weighed by the provided weights. For example, we could create weights based on the cross-validation error, that is, the lower the error, the larger the weight.

In *Step 8*, we created predictions using the blended models. To do so, we used the `predict_model` method and provided the blended model as the method’s argument. At this point, the `predict_model` method creates a forecast for the test set.

We also used the already familiar `plot_model` method to create a plot. When provided with a model, the `plot_model` method can display the model’s in-sample fit, the predictions on the test set, the out-of-sample predictions, or the model’s residuals.



Similar to the case of the `plot_model` method, we can also use the `check_stats` method together with the created model. When we pass the estimator, the method will perform the statistical tests on the model’s residuals.

In *Step 9*, we finalized the model using the `finalize_model` method. As we have seen in *Step 8*, the predictions we obtained were for the test set. In PyCaret's terminology, finalizing the model means that we take the model from the previous stages (without changing the selected hyperparameters) and then train the model using the entire dataset (both training and test sets). Having done so, we can create forecasts for the future.

After finalizing the model, we used the same `predict_model` and `plot_model` methods to create and plot the forecasts for the first 6 months of 2020 (which are outside of our dataset). While calling the methods, we passed the finalized model as the `estimator` argument.

There's more...

PyCaret is a very versatile library and we have only scratched the surface of what it offers. For brevity's sake, we only mention some of its features:

- Mature classification and regression AutoML capabilities. In this recipe, we have only used the time series module.
- Anomaly detection for time series.
- Integration with MLFlow for experiment logging.
- Using the time series module, we can easily train a single model instead of all of the available ones. We can do so using the `create_model` method. As the `estimator` argument, we need to pass the name of the model. We can get the names of the available models using the `models` method. Additionally, depending on the model we choose, we might want to pass some kwargs. For example, we might want to specify the order parameters of ARIMA models.
- As we have seen in the list of available models, except for the classic statistical models, PyCaret also offers selected ML models using the reduced regression approach. Those models also detrend and conditionally deseasonalize the time series to make it easier for the regression model to capture the autoregressive properties of the data.



You might also want to explore the `autots` library, which is another AutoML tool for time series forecasting.

Summary

In this chapter, we have covered the ML-based approaches to time series forecasting. We started with an extensive overview of validation approaches relevant to the time series domain. Furthermore, some of those were created to account for the intricacies of validating time series predictions in the financial domain.

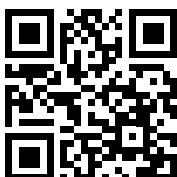
Then, we explored feature engineering and the concept of reduced regression, which allows us to use any regression algorithms for a time series forecasting task. Lastly, we covered Meta's Prophet algorithm and PyCaret—a low-code tool that automates machine learning workflows.

While exploring time series forecasting, we tried to introduce the most relevant Python libraries. However, there are quite a few other interesting positions worth mentioning. You can find some of them below:

- `autots`—AutoTS is an alternative AutoML library for time series forecasting.
- `darts`—Similar to `sktime`, it offers an entire framework for working with time series. The library contains a wide variety of models, starting with classic models such as ARIMA and ending with various popular neural network architectures used for time series forecasting.
- `greykite`—LinkedIn's Greykite library for time series forecasting, including its Silverkite algorithm.
- `kats`—A toolkit to analyze time series analysis developed by Meta. The library attempts to provide a one-stop shop for time series analysis, including tasks such as detection (for example, changepoint), forecasting, feature extraction, and more.
- `merlion`—Salesforce's ML library for time series analysis.
- `orbit`—Uber's library for Bayesian time series forecasting and inference.
- `statsforecast`—The library offers a collection of popular time series forecasting models (for example, autoARIMA and ETS), which are further optimized for high performance using `numba`.
- `stumpy`—A library that efficiently computes the matrix profile, which can be used for many time series-related tasks.
- `tslearn`—A toolkit for time series analysis.
- `tfp.sts`—A library in TensorFlow Probability used for forecasting using structural time series models.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

8

Multi-Factor Models

This chapter is devoted to estimating various factor models. **Factors** are variables/attributes that in the past were correlated with (then future) stock returns and are expected to contain the same predictive signals in the future.

These risk factors can be considered a tool for understanding the cross-section of (expected) returns. That is why various **factor models** are used to explain the excess returns (over the risk-free rate) of a certain portfolio or asset using one or more factors. We can think of the factors as the sources of risk that are the drivers of those excess returns. Each factor carries a risk premium and the overall portfolio/asset return is the weighted average of those premiums.

Factor models play a crucial role in portfolio management, mainly because:

- They can be used to identify interesting assets that can be added to the investment portfolio, which—in turn—should lead to better-performing portfolios.
- Estimating the exposure of a portfolio/asset to the factors allows for better risk management.
- We can use the models to assess the potential incremental value of adding a new risk factor.
- They make portfolio optimization easier, as summarizing the returns of many assets with a smaller number of factors reduces the amount of data required to estimate the covariance matrix.
- They can be used to assess a portfolio manager's performance—whether the performance (relative to the benchmark) is due to asset selection and timing of the trades, or if it comes from the exposure to known return drivers (factors).

By the end of this chapter, we will have constructed some of the most popular factor models. We will start with the simplest, yet very popular, one-factor model (which is the same as the **Capital Asset Pricing Model** when the considered factor is the market return) and then explain how to estimate more advanced three-, four-, and five-factor models. We will also cover the interpretation of what these factors represent and give a high-level overview of how they are constructed.

In this chapter, we cover the following recipes:

- Estimating the CAPM
- Estimating the Fama-French three-factor model
- Estimating the rolling three-factor model on a portfolio of assets
- Estimating the four- and five-factor models
- Estimating cross-sectional factor models using the Fama-MacBeth regression

Estimating the CAPM

In this recipe, we learn how to estimate the famous **Capital Asset Pricing Model (CAPM)** and obtain the beta coefficient. This model represents the relationship between the expected return on a risky asset and the market risk (also known as systematic or undiversifiable risk). CAPM can be considered a one-factor model, on top of which more complex factor models were built.

CAPM is represented by the following equation:

$$E(r_i) = r_f + \beta_i(E(r_m) - r_f)$$

Here, $E(r_i)$ denotes the expected return on asset i , r_f is the risk-free rate (such as a government bond), $E(r_m)$ is the expected return on the market, and β is the beta coefficient.

Beta can be interpreted as the level of the asset return's sensitivity, as compared to the market in general. Below we mention the possible interpretations of the coefficient:

- $\beta \leq -1$: The asset moves in the opposite direction to the benchmark and in a greater amount than the negative of the benchmark.
- $-1 < \beta < 0$: The asset moves in the opposite direction to the benchmark.
- $\beta = 0$: There is no correlation between the asset's price movement and the market benchmark.
- $0 < \beta < 1$: The asset moves in the same direction as the market, but the amount is smaller. An example might be the stock of a company that is not very susceptible to day-to-day fluctuations.
- $\beta = 1$: The asset and the market are moving in the same direction by the same amount.
- $\beta > 1$: The asset moves in the same direction as the market, but the amount is greater. An example might be the stock of a company that is very susceptible to day-to-day market news.

CAPM can also be represented as:

$$E(r_i) - r_f = \beta_i(E(r_m) - r_f)$$

In this specification, the left-hand side of the equation can be interpreted as the risk premium, while the right-hand side contains the market premium. The same equation can be further reshaped into:

$$\beta = \frac{\text{cov}(R_i, R_m)}{\text{var}(R_m)}$$

Here, $R_i = E(r_i) - r_f$ and $R_m = E(r_m) - r_f$.

In this example, we consider the case of Amazon and assume that the S&P 500 index represents the market. We use 5 years (2016 to 2020) of monthly data to estimate the beta. In current times, the risk-free rate is so low that, for simplicity's sake, we assume it is equal to zero.

How to do it...

Execute the following steps to implement the CAPM in Python:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
import statsmodels.api as sm
```

2. Specify the risky asset, the benchmark, and the time horizon:

```
RISKY_ASSET = "AMZN"
MARKET_BENCHMARK = "^GSPC"
START_DATE = "2016-01-01"
END_DATE = "2020-12-31"
```

3. Download the necessary data from Yahoo Finance:

```
df = yf.download([RISKY_ASSET, MARKET_BENCHMARK],
                 start=START_DATE,
                 end=END_DATE,
                 adjusted=True,
                 progress=False)
```

4. Resample to monthly data and calculate the simple returns:

```
X = (
    df["Adj Close"]
    .rename(columns={RISKY_ASSET: "asset",
                    MARKET_BENCHMARK: "market"})
    .resample("M")
    .last()
    .pct_change()
    .dropna()
)
```

5. Calculate beta using the covariance approach:

```
covariance = X.cov().iloc[0,1]
benchmark_variance = X.market.var()
beta = covariance / benchmark_variance
```

The result of the code is `beta = 1.2035`.

6. Prepare the input and estimate the CAPM as a linear regression:

```
# separate target
y = X.pop("asset")

# add constant
X = sm.add_constant(X)

# define and fit the regression model
capm_model = sm.OLS(y, X).fit()

# print results
print(capm_model.summary())
```

Figure 8.1 shows the results of estimating the CAPM model:

OLS Regression Results										
Dep. Variable:	asset	R-squared:	0.408							
Model:	OLS	Adj. R-squared:	0.398							
Method:	Least Squares	F-statistic:	40.05							
Date:	Wed, 02 Mar 2022	Prob (F-statistic):	3.89e-08							
Time:	23:28:10	Log-Likelihood:	80.639							
No. Observations:	60	AIC:	-157.3							
Df Residuals:	58	BIC:	-153.1							
Df Model:	1									
Covariance Type:	nonrobust									
	coef	std err	t	P> t	[0.025	0.975]				
const	0.0167	0.009	1.953	0.056	-0.000	0.034				
market	1.2035	0.190	6.329	0.000	0.823	1.584				
Omnibus:	2.202	Durbin-Watson:	1.783							
Prob(Omnibus):	0.333	Jarque-Bera (JB):	1.814							
Skew:	0.426	Prob(JB):	0.404							
Kurtosis:	2.989	Cond. No.	23.0							

Figure 8.1: The summary of the CAPM estimated using OLS

These results indicate that the beta (denoted as the market here) is equal to 1.2, which means that Amazon's returns are 20% more volatile than the market (proxied by S&P 500). Or in other words, Amazon's (excess) return is expected to move 1.2 times the market (excess) return. The value of the intercept is relatively small and statistically insignificant at the 5% significance level.

How it works...

First, we specified the assets we wanted to use (Amazon and S&P 500) and the time frame. In *Step 3*, we downloaded the data from Yahoo Finance. Then, we only kept the last available price per month and calculated the monthly returns as the percentage change between the subsequent observations.

In *Step 5*, we calculated the beta as the ratio of the covariance between the risky asset and the benchmark to the benchmark's variance.

In *Step 6*, we separated the target (Amazon's stock returns) and the features (S&P 500 returns) using the `pop` method of a pandas DataFrame. Afterward, we added the constant to the features (effectively adding a column of ones to the DataFrame) with the `add_constant` function.

The idea behind adding the intercept to this regression is to investigate whether—after estimating the model—the intercept (in the case of the CAPM, also known as **Jensen's alpha**) is zero. If it is positive and significant, it means that—assuming the CAPM model is true—the asset or portfolio generates abnormally high risk-adjusted returns. There are two possible implications: either the market is inefficient or there are some other undiscovered risk factors that should be included in the model. This issue is known as the **joint hypothesis problem**.



We can also use the formula notation, which adds the constant automatically. To do so, we must import `statsmodels.formula.api` as `smf` and then run the slightly modified line: `capm_model = smf.ols(formula="asset ~ market", data=X).fit()`. The results of both approaches are the same. You can find the complete code in the accompanying Jupyter notebook.

Lastly, we ran the OLS regression and printed the summary. Here, we could see that the coefficient by the market variable (that is, the CAPM beta) is equal to the beta that was calculated using the covariance between the asset and the market in *Step 5*.

There's more...

In the example above, we assumed there was no risk-free rate, which is a reasonable assumption to make nowadays. However, there might be cases when we would like to account for a non-zero risk-free rate. To do so, we could use one of the following approaches.

Using data from Prof. Kenneth French's website

The market premium ($r_m - r_f$) and the risk-free rate (approximated by the one-month Treasury Bill) can be downloaded from Professor Kenneth French's website (please refer to the *See also* section of this recipe for the link).



Please bear in mind that the definition of the market benchmark used by Prof. French is different from the S&P 500 index—a detailed description is available on his website. For a description of how to easily download the data, please refer to the *Implementing the Fama-French three-factor model* recipe.

Using the 13-Week T-bill

The second option is to approximate the risk-free rate with, for example, the 13-Week (3-month) Treasury Bill (Yahoo Finance ticker: ^IRX).

Follow these steps to learn how to download the data and convert it into the appropriate risk-free rate:

1. Define the length of the period in days:

```
N_DAYS = 90
```

2. Download the data from Yahoo Finance:

```
df_rf = yf.download("^IRX",
                    start=START_DATE,
                    end=END_DATE,
                    progress=False)
```

3. Resample the data to monthly frequency (by taking the last value for each month):

```
rf = df_rf.resample("M").last().Close / 100
```

4. Calculate the risk-free return (expressed as daily values) and convert the values to monthly:

```
rf = ( 1 / (1 - rf * N_DAYS / 360) )**(1 / N_DAYS)
rf = (rf ** 30) - 1
```

5. Plot the calculated risk-free rate:

```
rf.plot(title="Risk-free rate (13-Week Treasury Bill)")
```

Figure 8.2 shows the visualization of the risk-free rate over time:

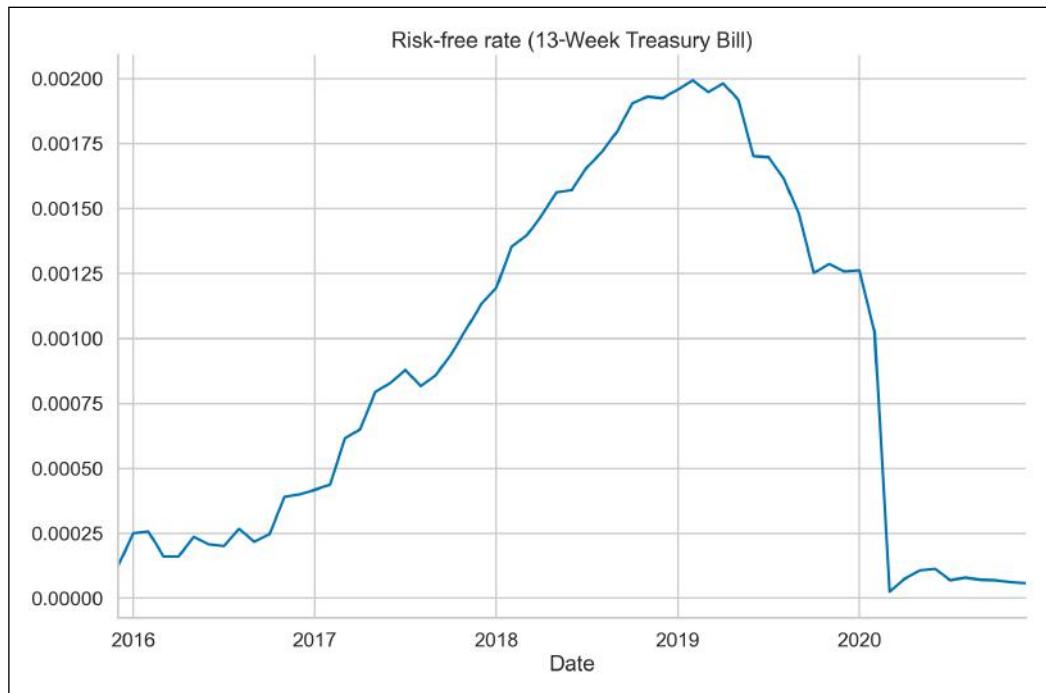


Figure 8.2: The risk-free rate calculated using the 13-Week Treasury Bill

Using the 3-Month T-bill from the FRED database

The last approach is to approximate the risk-free rate using the 3-Month Treasury Bill (Secondary Market Rate), which can be downloaded from the **Federal Reserve Economic Data (FRED)** database.

Follow these steps to learn how to download the data and convert it to a monthly risk-free rate:

1. Import the library:

```
import pandas_datareader.data as web
```

2. Download the data from the FRED database:

```
rf = web.DataReader(
    "TB3MS", "fred", start=START_DATE, end=END_DATE
)
```

3. Convert the obtained risk-free rate to monthly values:

```
rf = (1 + (rf / 100)) ** (1 / 12) - 1
```

4. Plot the calculated risk-free rate:

```
rf.plot(title="Risk-free rate (3-Month Treasury Bill)")
```

We can compare the results of the two methods by comparing the plots of the risk-free rates:

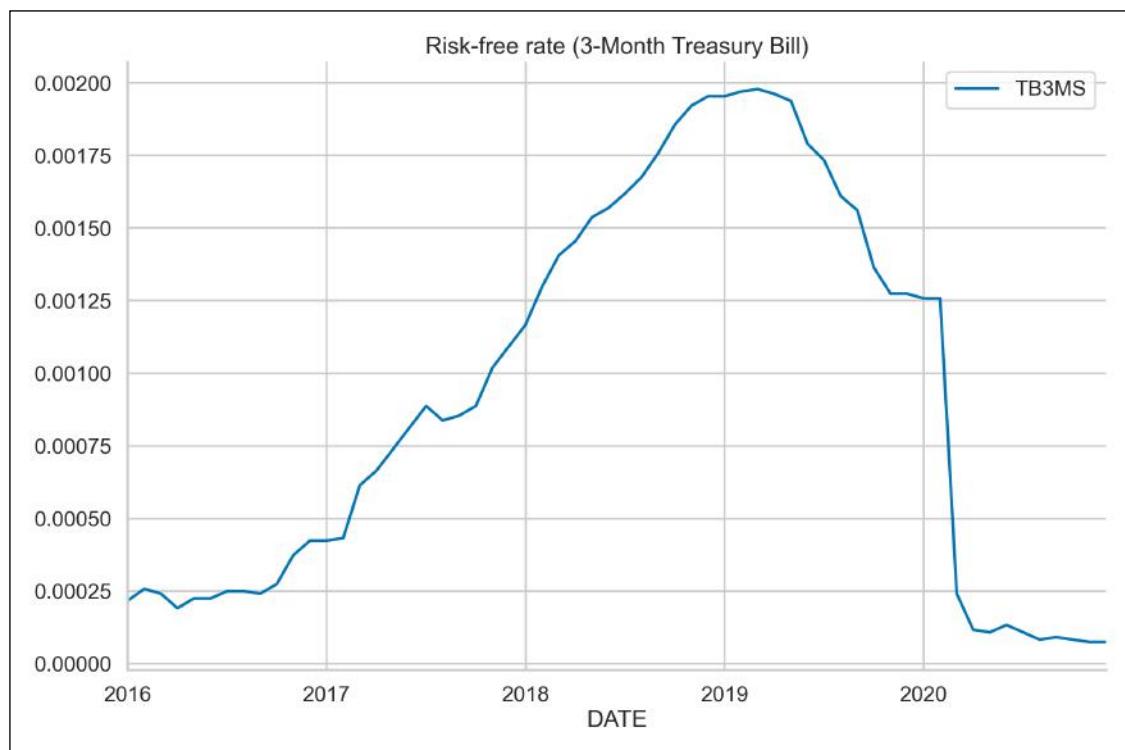


Figure 8.3: The risk-free rate calculated using the 3-Month Treasury Bill

The above lets us conclude that the plots look very similar.

See also

Additional resources are available here:

- Sharpe, W. F., “Capital asset prices: A theory of market equilibrium under conditions of risk,” *The Journal of Finance*, 19, 3 (1964): 425–442.
- Risk-free rate data on Prof. Kenneth French’s website: http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/F-F_Research_Data_Factors_CSV.zip.

Estimating the Fama-French three-factor model

In their famous paper, Fama and French expanded the CAPM model by adding two additional factors explaining the excess returns of an asset or portfolio. The factors they considered are:

- **The market factor (MKT):** It measures the excess return of the market, analogical to the one in the CAPM.
- **The size factor (SMB; Small Minus Big):** It measures the excess return of stocks with a small market cap over those with a large market cap.
- **The value factor (HML; High Minus Low):** It measures the excess return of value stocks over growth stocks. Value stocks have a high book-to-market ratio, while growth stocks are characterized by a low ratio.



Please see the *See also* section for a reference to how the factors are calculated.

The model can be represented as follows:

$$E(r_i) = r_f + \alpha + \beta_{mkt}(E(r_m) - r_f) + \beta_{smb}SMB + \beta_{hml}HML$$

Or in its simpler form:

$$E(r_i) - r_f = \alpha + \beta_{mkt}MKT + \beta_{smb}SMB + \beta_{hml}HML$$

Here, $E(r_i)$ denotes the expected return on asset i , r_f is the risk-free rate (such as a government bond), and α is the intercept. The reason for including the intercept is to make sure its value is equal to 0. This confirms that the three-factor model correctly evaluates the relationship between the excess returns and the factors.



In the case of a statistically significant, non-zero intercept, the model might not evaluate the asset/portfolio return correctly. However, the authors stated that the three-factor model is “fairly correct,” even when it is unable to pass the statistical test.

Due to the popularity of this approach, these factors became collectively known as the **Fama-French Factors** or the **Three-Factor Model**. They have been widely accepted in both academia and the industry as stock market benchmarks and they are often used to evaluate investment performance.

In this recipe, we estimate the three-factor model using 5 years (2016 to 2020) of monthly returns on Apple's stock.

How to do it...

Follow these steps to implement the three-factor model in Python:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
import statsmodels.formula.api as smf
import pandas_datareader.data as web
```

2. Define the parameters:

```
RISKY_ASSET = "AAPL"
START_DATE = "2016-01-01"
END_DATE = "2020-12-31"
```

3. Download the dataset containing the risk factors:

```
ff_dict = web.DataReader("F-F_Research_Data_Factors",
                        "famafrench",
                        start=START_DATE,
                        end=END_DATE)
```

The downloaded dictionary contains three elements: the monthly factors from the requested time frame (indexed as 0), the corresponding annual factors (indexed as 1), and a short description of the dataset (indexed as DESCRIPTOR).

4. Select the appropriate dataset and divide the values by 100:

```
factor_3_df = ff_dict[0].rename(columns={"Mkt-RF": "MKT"}) \
              .div(100)
factor_3_df.head()
```

The resulting data should look as follows:

	MKT	SMB	HML	RF
Date				
2016-01	-0.0577	-0.0339	0.0207	0.0001
2016-02	-0.0008	0.0081	-0.0057	0.0002
2016-03	0.0696	0.0075	0.0110	0.0002
2016-04	0.0092	0.0067	0.0321	0.0001
2016-05	0.0178	-0.0019	-0.0165	0.0001

Figure 8.4: Preview of the downloaded factors

- Download the prices of the risky asset:

```
asset_df = yf.download(RISKY_ASSET,
                      start=START_DATE,
                      end=END_DATE,
                      adjusted=True)
```

- Calculate the monthly returns on the risky asset:

```
y = asset_df["Adj Close"].resample("M") \
    .last() \
    .pct_change() \
    .dropna()

y.index = y.index.to_period("m")
y.name = " rtn"
```

- Merge the datasets and calculate the excess returns:

```
factor_3_df = factor_3_df.join(y)
factor_3_df["excess_rtn"] = (
    factor_3_df[" rtn"] - factor_3_df["RF"]
)
```

- Estimate the three-factor model:

```
ff_model = smf.ols(formula="excess_rtn ~ MKT + SMB + HML",
                    data=factor_3_df).fit()
print(ff_model.summary())
```

The results of the three-factor model are presented below:

OLS Regression Results									
Dep. Variable:	excess_rtn	R-squared:	0.504						
Model:	OLS	Adj. R-squared:	0.477						
Method:	Least Squares	F-statistic:	18.94						
Date:	Wed, 02 Mar 2022	Prob (F-statistic):	1.32e-08						
Time:	23:48:12	Log-Likelihood:	82.679						
No. Observations:	60	AIC:	-157.4						
Df Residuals:	56	BIC:	-149.0						
Df Model:	3								
Covariance Type:	nonrobust								
	coef	std err	t	P> t	[0.025	0.975]			
Intercept	0.0084	0.009	0.954	0.344	-0.009	0.026			
MKT	1.4264	0.198	7.213	0.000	1.030	1.823			
SMB	-0.4590	0.359	-1.280	0.206	-1.177	0.259			
HML	-0.7186	0.260	-2.759	0.008	-1.240	-0.197			
Omnibus:		8.642	Durbin-Watson:		2.458				
Prob(Omnibus):		0.013	Jarque-Bera (JB):		8.952				
Skew:		-0.652	Prob(JB):		0.0114				
Kurtosis:		4.371	Cond. No.		45.8				

Figure 8.5: The summary of the estimated three-factor model

When interpreting the results of the three-factor model, we should pay attention to two issues:

- Whether the intercept is positive and statistically significant
- Which factors are statistically significant and if their direction matches past results (for example, based on a literature study) or our assumptions

In our case, the intercept is positive, but not statistically significant at the 5% significance level. Of the risk factors, only the SMB factor is not significant. However, a thorough literature study is required to formulate a hypothesis about the factors and their direction of influence.

We can also look at the F-statistic that was presented in the regression summary, which tests the joint significance of the regression. The null hypothesis states that coefficients of all features (factors, in this case), except for the intercept, have values equal to 0. We can see that the corresponding p -value is much lower than 0.05, which gives us reason to reject the null hypothesis at the 5% significance level.

How it works...

In the first two steps, we imported the required libraries and defined the parameters – the risky asset (Apple's stock) and the considered time frame.

In *Step 3*, we downloaded the data using the functionality of the `pandas_datareader` library. We had to specify which dataset (see the *There's more...* section for information on inspecting the available datasets) and reader (`famafrench`) we wanted to use, as well as the start/end dates (by default, `web.DataReader` downloads the last 5 years' worth of data).

In *Step 4*, we selected only the dataset containing monthly values (indexed as `0` in the downloaded dictionary), renamed the column containing the `MKT` factor, and divided all the values by 100. We did it to arrive at the correct encoding of percentages; for example, a value of 3.45 in the dataset represents 3.45%.

In *Steps 5* and *6*, we downloaded and wrangled the prices of Apple's stock. We obtained the monthly returns by calculating the percentage change of the end-of-month prices. In *Step 6*, we also changed the formatting of the index to `%Y-%m` (for example, `2000-12`) since the Fama-French factors contain dates in such a format. Then, we joined the two datasets in *Step 7*.

Finally, in *Step 8*, we ran the regression using the formula notation—we do not need to manually add an intercept when doing so. One thing worth mentioning is that the coefficient by the `MKT` variable will not be equal to the CAPM's beta, as there are also other factors in the model, and the factors' influence on the excess returns is distributed differently.

There's more...

We can use the following snippet to see what datasets from the Fama-French category are available for download using `pandas_datareader`. For brevity, we only display 5 of the approximately 300 available datasets:

```
from pandas_datareader.famafrench import get_available_datasets  
get_available_datasets()[:5]
```

Running the snippet returns the following list:

```
['F-F_Research_Data_Factors',  
'F-F_Research_Data_Factors_weekly',  
'F-F_Research_Data_Factors_daily',  
'F-F_Research_Data_5_Factors_2x3',  
'F-F_Research_Data_5_Factors_2x3_daily']
```

In the previous edition of the book, we also showed how to directly download the CSV files from Prof. French's website using simple Bash commands from within a Jupyter notebook. You can find the code explaining how to do that in the accompanying notebook.

See also

Additional resources:

- For details on how all the factors were calculated, please refer to Prof. French's website at http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/f-f_factors.html
- Fama, E. F., and French, K. R., "Common risk factors in the returns on stocks and bonds," Journal of Financial Economics, 33, 1 (1993): 3-56

Estimating the rolling three-factor model on a portfolio of assets

In this recipe, we learn how to estimate the three-factor model in a rolling fashion. What we mean by rolling is that we always consider an estimation window of a constant size (60 months, in this case) and roll it through the entire dataset, one period at a time. A potential reason for doing such an experiment is to test the stability of the results. Alternatively, we could also use an expanding window for this exercise.

In contrast to the previous recipes, this time, we use portfolio returns instead of a single asset. To keep things simple, we assume that our allocation strategy is to have an equal share of the total portfolio's value in each of the following stocks: Amazon, Google, Apple, and Microsoft. For this experiment, we use stock prices from the years 2010 to 2020.

How to do it...

Follow these steps to implement the rolling three-factor model in Python:

1. Import the libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
import statsmodels.formula.api as smf
import pandas_datareader.data as web
```

2. Define the parameters:

```
ASSETS = ["AMZN", "GOOG", "AAPL", "MSFT"]
WEIGHTS = [0.25, 0.25, 0.25, 0.25]
START_DATE = "2010-01-01"
END_DATE = "2020-12-31"
```

3. Download the factor-related data:

```
factor_3_df = web.DataReader("F-F_Research_Data_Factors",
                             "famafrench",
                             start=START_DATE,
                             end=END_DATE)[0]
factor_3_df = factor_3_df.div(100)
```

4. Download the prices of risky assets from Yahoo Finance:

```
asset_df = yf.download(ASSETS,
                      start=START_DATE,
                      end=END_DATE,
                      adjusted=True,
                      progress=False)
```

5. Calculate the monthly returns on the risky assets:

```
asset_df = asset_df["Adj Close"].resample("M") \
            .last() \
            .pct_change() \
            .dropna()
asset_df.index = asset_df.index.to_period("m")
```

6. Calculate the portfolio returns:

```
asset_df["portfolio_returns"] = np.matmul(
    asset_df[ASSETS].values, WEIGHTS
)
```

7. Merge the datasets:

```
factor_3_df = asset_df.join(factor_3_df).drop(ASSETS, axis=1)
factor_3_df.columns = ["portf rtn", "mkt", "smb", "hml", "rf"]
factor_3_df["portf_ex_rtn"] = (
    factor_3_df["portf rtn"] - factor_3_df["rf"]
)
```

8. Define a function for the rolling n -factor model:

```
def rolling_factor_model(input_data, formula, window_size):  
  
    coeffs = []  
    for start_ind in range(len(input_data) - window_size + 1):  
        end_ind = start_ind + window_size  
  
        ff_model = smf.ols(  
            formula=formula,  
            data=input_data[start_ind:end_ind]  
        ).fit()  
  
        coeffs.append(ff_model.params)  
  
    coeffs_df = pd.DataFrame(  
        coeffs,  
        index=input_data.index[window_size - 1:]  
    )  
  
    return coeffs_df
```



For a version with a docstring explaining the input/output, please refer to this book's GitHub repository.

9. Estimate the rolling three-factor model and plot the results:

```
MODEL_FORMULA = "portf_ex_rtn ~ mkt + smb + hml"  
results_df = rolling_factor_model(factor_3_df,  
                                  MODEL_FORMULA,  
                                  window_size=60)  
(  
    results_df  
    .plot(title = "Rolling Fama-French Three-Factor model",  
          style=[ "-", "--", "-.", ":" ])  
    .legend(loc="center left", bbox_to_anchor=(1.0, 0.5))  
)
```

Executing the code results in the following plot:

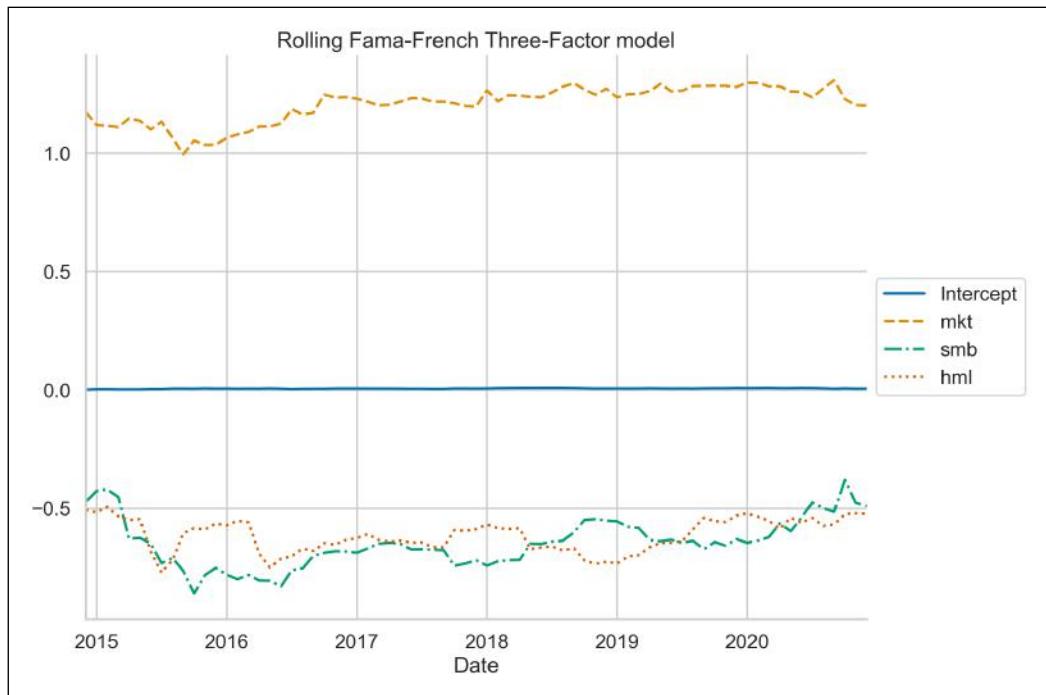


Figure 8.6: The coefficients of the rolling three-factor model

By inspecting the preceding plot, we can see the following:

- The intercept is almost constant and very close to 0.
- There is some variability in the factors, but no sudden reversals or unexpected jumps.

How it works...

In *Steps 3 and 4*, we downloaded data using `pandas_datareader` and `yfinance`. This is very similar to what we did in the *Estimating the Fama-French three-factor model* recipe, so at this point we will not go into too much detail about this.

In *Step 6*, we calculated the portfolio returns as a weighted average of the returns of the portfolio constituents (calculated in *Step 5*). This is possible as we are working with simple returns—for more details, please refer to the *Converting prices to returns* recipe in *Chapter 2, Data Preprocessing*. Bear in mind that this simple approach assumes that, at the end of each month, we have exactly the same asset allocation (as indicated by the weights). This can be achieved with **portfolio rebalancing**, that is, adjusting the allocation after a specified period of time to always match the intended weights' distribution.

Afterward, we merged the two datasets in *Step 7*. In *Step 8*, we defined a function for estimating the n -factor model using a rolling window. The main idea is to loop over the DataFrame we prepared in previous steps and for each month, estimate the Fama-French model using the last 5 years' worth of data (60 months). By appropriately slicing the input DataFrame, we made sure that we only estimate the model from the 60th month onward, to make sure we always have a full window of observations.



Proper software engineering best practices would suggest writing some assertions to make sure the types of the inputs are as we intended them to be, or that the input DataFrame contains the necessary columns. However, we have not done this here for brevity.

Finally, we applied the defined function to the prepared DataFrame and plotted the results.

Estimating the four- and five-factor models

In this recipe, we implement two extensions of the Fama-French three-factor model.

First, **Carhart's four-factor model**: The underlying assumption of this extension is that, within a short period of time, a winner stock will remain a winner, while a loser will remain a loser. An example of a criterion for classifying winners and losers could be the last 12-month cumulative total returns. After identifying the two groups, we long the winners and short the losers within a certain holding period.

The **momentum factor** (WML; Winners Minus Losers) measures the excess returns of the winner stocks over the loser stocks in the past 12 months (please refer to the *See also* section of this recipe for references on the calculations of the momentum factor).

The four-factor model can be expressed as follows:

$$E(r_i) - r_f = \alpha + \beta_{mkt}MKT + \beta_{smb}SMB + \beta_{hml}HML + \beta_{wml}WML$$

The second extension is **Fama-French's five-factor model**. Fama and French expanded their three-factor model by adding two factors:

- **The profitability factor** (RMW; Robust Minus Weak) measures the excess returns of companies with high profit margins (robust profitability) over those with lower profits (weak profitability).
- **The investment factor** (CMA; Conservative Minus Aggressive) measures the excess returns of firms with low investment policies (conservative) over those investing more (aggressive).

The five-factor model can be expressed as follows:

$$E(r_i) - r_f = \alpha + \beta_{mkt}MKT + \beta_{smb}SMB + \beta_{hml}HML + \beta_{rmw}RMW + \beta_{cma}CMA$$

Like in all factor models, if the exposure to the risk factors captures all possible variations in expected returns, the intercept (α) for all the assets/portfolios should be equal to zero.

In this recipe, we explain monthly returns on Amazon from 2016 to 2020 with the four- and five-factor models.

How to do it...

Follow these steps to implement the four- and five-factor models in Python:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
import statsmodels.formula.api as smf
import pandas_datareader.data as web
```

2. Specify the risky asset and the time horizon:

```
RISKY_ASSET = "AMZN"
START_DATE = "2016-01-01"
END_DATE = "2020-12-31"
```

3. Download the risk factors from Prof. French's website:

```
# three factors
factor_3_df = web.DataReader("F-F_Research_Data_Factors",
                             "famafrench",
                             start=START_DATE,
                             end=END_DATE)[0]

# momentum factor
momentum_df = web.DataReader("F-F_Momentum_Factor",
                             "famafrench",
                             start=START_DATE,
                             end=END_DATE)[0]

# five factors
factor_5_df = web.DataReader("F-F_Research_Data_5_Factors_2x3",
                             "famafrench",
                             start=START_DATE,
                             end=END_DATE)[0]
```

4. Download the data of the risky asset from Yahoo Finance:

```
asset_df = yf.download(RISKY_ASSET,
                      start=START_DATE,
                      end=END_DATE,
                      adjusted=True,
                      progress=False)
```

5. Calculate the monthly returns:

```
y = asset_df["Adj Close"].resample("M") \
    .last() \
    .pct_change() \
    .dropna()

y.index = y.index.to_period("m")
y.name = "rtn"
```

6. Merge the datasets for the four-factor model:

```
# join all datasets on the index
factor_4_df = factor_3_df.join(momentum_df).join(y)

# rename columns
factor_4_df.columns = ["mkt", "smb", "hml", "rf", "mom", "rtn"]

# divide everything (except returns) by 100
factor_4_df.loc[:, factor_4_df.columns != "rtn"] /= 100

# calculate excess returns
factor_4_df["excess_rtn"] = (
    factor_4_df["rtn"] - factor_4_df["rf"]
)
```

7. Merge the datasets for the five-factor model:

```
# join all datasets on the index
factor_5_df = factor_5_df.join(y)

# rename columns
factor_5_df.columns = [
    "mkt", "smb", "hml", "rmw", "cma", "rf", "rtn"
]

# divide everything (except returns) by 100
factor_5_df.loc[:, factor_5_df.columns != "rtn"] /= 100

# calculate excess returns
factor_5_df["excess_rtn"] = (
    factor_5_df["rtn"] - factor_5_df["rf"]
)
```

8. Estimate the four-factor model:

```
four_factor_model = smf.ols(
    formula="excess_rtn ~ mkt + smb + hml + mom",
    data=factor_4_df
).fit()

print(four_factor_model.summary())
```

Figure 8.7 shows the results:

OLS Regression Results									
Dep. Variable:	excess_rtn	R-squared:	0.563						
Model:	OLS	Adj. R-squared:	0.532						
Method:	Least Squares	F-statistic:	17.74						
Date:	Thu, 03 Mar 2022	Prob (F-statistic):	2.10e-09						
Time:	00:07:34	Log-Likelihood:	89.673						
No. Observations:	60	AIC:	-169.3						
Df Residuals:	55	BIC:	-158.9						
Df Model:	4								
Covariance Type:	nonrobust								
	coef	std err	t	P> t	[0.025	0.975]			
Intercept	0.0054	0.008	0.676	0.502	-0.011	0.021			
mkt	1.4461	0.188	7.709	0.000	1.070	1.822			
smb	-0.4336	0.340	-1.276	0.207	-1.115	0.247			
hml	-0.7914	0.274	-2.888	0.006	-1.341	-0.242			
mom	0.2220	0.269	0.826	0.412	-0.316	0.760			
Omnibus:	0.390	Durbin-Watson:	2.032						
Prob(Omnibus):	0.823	Jarque-Bera (JB):	0.276						
Skew:	0.163	Prob(JB):	0.871						
Kurtosis:	2.933	Cond. No.	50.8						

Figure 8.7: The summary of the estimated four-factor model

9. Estimate the five-factor model:

```
five_factor_model = smf.ols(
    formula="excess_rtn ~ mkt + smb + hml + rmw + cma",
    data=factor_5_df
).fit()

print(five_factor_model.summary())
```

Figure 8.8 shows the results:

OLS Regression Results									
Dep. Variable:	excess_rtn	R-squared:	0.612						
Model:	OLS	Adj. R-squared:	0.576						
Method:	Least Squares	F-statistic:	17.01						
Date:	Thu, 03 Mar 2022	Prob (F-statistic):	4.54e-10						
Time:	00:07:35	Log-Likelihood:	93.191						
No. Observations:	60	AIC:	-174.4						
Df Residuals:	54	BIC:	-161.8						
Df Model:	5								
Covariance Type:	nonrobust								
	coef	std err	t	P> t	[0.025	0.975]			
Intercept	0.0059	0.008	0.772	0.444	-0.009	0.021			
mkt	1.5117	0.193	7.851	0.000	1.126	1.898			
smb	-0.9411	0.335	-2.807	0.007	-1.613	-0.269			
hml	-0.5433	0.281	-1.936	0.058	-1.106	0.019			
rmw	-1.1628	0.513	-2.266	0.027	-2.191	-0.134			
cma	-0.5153	0.509	-1.012	0.316	-1.536	0.505			
Omnibus:		0.073	Durbin-Watson:		2.074				
Prob(Omnibus):		0.964	Jarque-Bera (JB):		0.181				
Skew:		-0.077	Prob(JB):		0.913				
Kurtosis:		2.779	Cond. No.		79.4				

Figure 8.8: The summary of the estimated five-factor model

According to the five-factor model, Amazon's excess returns are negatively exposed to most of the factors (all but the market factor). Here, we present an example of the interpretation of the coefficients: an increase by 1 percentage point in the market factor results in an increase of 0.015 p.p. In other words, for a 1% return by the market factor, we can expect our portfolio (Amazon's stock) to return $1.5117 * 1\%$ in excess of the risk-free rate.

Similar to the three-factor model, if the five-factor model fully explains the excess stock returns, the estimated intercept should be statistically indistinguishable from zero (which is the case for the considered problem).

How it works...

In *Step 2*, we defined the parameters—the ticker of the considered stock and timeframes.

In *Step 3*, we downloaded the necessary datasets using `pandas_datareader`, which provides us with a convenient way of downloading the risk factor-related data without manually downloading the CSV files. For more information on this process, please refer to the *Estimating the Fama-French three-factor model* recipe.

In *Steps 4* and *5*, we downloaded Amazon's stock prices and calculated the monthly returns using the previously explained methodology.

In *Steps 6* and *7*, we joined all the datasets, renamed the columns, and calculated the excess returns. When using the `join` method without specifying what we want to join on (the `on` argument), the default is the index of the DataFrames.

This way, we prepared all the necessary inputs for the four- and five-factor models. We also had to divide all the data we downloaded from Prof. French's website by 100 to arrive at the correct scale.



SMB factor in the five-factor dataset is calculated differently compared to how it is in the three-factor dataset. For more details, please refer to the link in the *See also* section of this recipe.

In *Step 8* and *Step 9*, we estimated the models using the functional form of OLS regression from the `statsmodels` library. The functional form automatically adds the intercept to the regression equation.

See also

For details on the calculation of the factors, please refer to the following links:

- Momentum factor: https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/det_mom_factor.html
- Five-factor model: https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/Data_Library/f-f_5_factors_2x3.html

For papers introducing the four- and five-factor models, please refer to the following links:

- Carhart, M. M. (1997), “*On Persistence in Mutual Fund Performance*,” *The Journal of Finance*, 52, 1 (1997): 57-82
- Fama, E. F. and French, K. R. 2015. “A five-factor asset pricing model,” *Journal of Financial Economics*, 116(1): 1-22: <https://doi.org/10.1016/j.jfineco.2014.10.010>

Estimating cross-sectional factor models using the Fama-MacBeth regression

In the previous recipes, we have covered estimating different factor models using a single asset or portfolio as the dependent variable. However, we can estimate the factor models for multiple assets at once, using cross-section (panel) data.

Following this approach, we can:

- Estimate the portfolios' exposure to the risk factors and learn how much those factors drive the portfolios' returns
- Understand how much taking a given risk is worth by knowing the premium that the market pays for the exposure to a certain factor

Knowing the risk premiums, we can then estimate the returns for any portfolio provided we can approximate that portfolio's exposure to the risk factors.

While estimating cross-sectional regression, we can encounter multiple problems due to the fact that some assumptions of linear regression might not hold. We might encounter the following:

- Heteroskedasticity and serial correlation, leading to the covariation of residuals
- Multicollinearity
- Measurement errors

To solve those issues, we can use a technique called the **Fama-MacBeth regression**, which is a two-step procedure specifically designed to estimate the premiums rewarded by the market for the exposure to certain risk factors.

The steps are as follows:

1. Obtain the factor loadings by estimating N (the number of portfolios/assets) time-series regressions of excess returns on the factors:

$$r_i = F * \beta_i + \varepsilon_i$$

2. Obtain the risk premiums by estimating T (the number of periods) cross-sectional regressions, one for each period:

$$r_t = \hat{\beta} * \lambda_t$$

In this recipe, we estimate the Fama-MacBeth regression using five risk factors and the returns of 12 industry portfolios, also available on Prof. French's website.

How to do it...

Execute the following steps to estimate the Fama-MacBeth regression:

1. Import the libraries:

```
import pandas as pd
import pandas_datareader.data as web
from linearmodels.asset_pricing import LinearFactorModel
```

2. Specify the time horizon:

```
START_DATE = "2010"
END_DATE = "2020-12"
```

3. Download and adjust the risk factors from Prof. French's website:

```
factor_5_df = (
    web.DataReader("F-F_Research_Data_5_Factors_2x3",
                  "famafrench",
                  start=START_DATE,
                  end=END_DATE)[0]
    .div(100)
)
```

4. Download and adjust the returns of 12 Industry Portfolios from Prof. French's website:

```
portfolio_df = (
    web.DataReader("12_Industry_Portfolios",
                  "famafrench",
                  start=START_DATE,
                  end=END_DATE)[0]
    .div(100)
    .sub(factor_5_df["RF"], axis=0)
)
```

5. Drop the risk-free rate from the factor dataset:

```
factor_5_df = factor_5_df.drop("RF", axis=1)
```

6. Estimate the Fama-MacBeth regression and print the summary:

```
five_factor_model = LinearFactorModel(
    portfolios=portfolio_df,
    factors=factor_5_df
)
result = five_factor_model.fit()
print(result)
```

Running the snippet generates the following summary:

LinearFactorModel Estimation Summary						
No. Test Portfolios:	12	R-squared:	0.7906			
No. Factors:	5	J-statistic:	9.9132			
No. Observations:	132	P-value	0.1935			
Date:	Thu, Mar 03 2022	Distribution:	chi2(7)			
Time:	00:14:16					
Cov. Estimator:	robust					
Risk Premia Estimates						
	Parameter	Std. Err.	T-stat	P-value	Lower CI	Upper CI
Mkt-RF	0.0123	0.0038	3.2629	0.0011	0.0049	0.0198
SMB	-0.0063	0.0052	-1.2085	0.2269	-0.0165	0.0039
HML	-0.0089	0.0032	-2.7764	0.0055	-0.0152	-0.0026
RMW	-0.0009	0.0046	-0.1953	0.8451	-0.0099	0.0081
CMA	-0.0025	0.0039	-0.6467	0.5178	-0.0100	0.0051

Figure 8.9: The results of the Fama-MacBeth regression

The results in the table are average risk premiums from the T cross-sectional regressions.

We can also print the full summary (containing the risk premiums and each portfolio's factor loadings). To do so, we need to run the following line of code:

```
print(result.full_summary)
```

How it works...

In the first two steps, we imported the required libraries and defined the start and end date of our exercise. In total, we will be using 11 years of monthly data, resulting in 132 observations of the variables (denoted as T). For the end date, we had to specify 2020-12. Using 2020 alone would result in the downloaded datasets ending with January 2020.

In *Step 3*, we downloaded the five-factor data set using `pandas_datareader`. We adjusted the values to express percentages by dividing them by 100.

In *Step 4*, we downloaded the returns on 12 Industry Portfolios from Prof. French's website (please see the link in the *See also* section for more details on the dataset). We have also adjusted the values by dividing them by 100 and calculated the excess returns by subtracting the risk-free rate (available at the factor dataset) from each column of the portfolio dataset. We could do that easily using the `sub` method as the time periods are an exact match.

In Step 5, we dropped the risk-free rate, as we will not be using it anymore and it will be easier to estimate the Fama-MacBeth regression model with no redundant columns in the DataFrames.

In the last step, we instantiated an object of the `LinearFactorModel` class and provided both datasets as arguments. Then, we used the `fit` method to estimate the model. Lastly, we printed the summary.



You might notice a small difference between `linearmodels` and `scikit-learn`. In the latter, we provide the data while calling the `fit` method. With `linearmodels`, we had to provide the data while creating an instance of the `LinearFactorModel` class.



In `linearmodels`, you can also use the formula notation (as we have done when estimating the factor models using `statsmodels`). To do so, we need to use the `from_formula` method. An example could look as follows:

`LinearFactorModel.from_formula(formula, data)`, where `formula` is the string containing the formula and `data` is an object containing both the portfolios/assets and the factors.

There's more...

We have already estimated the Fama-MacBeth regression using the `linearmodels` library. However, it might strengthen our understanding of the procedure to carry out the two steps manually.

Execute the following steps to carry out the two steps of the Fama-MacBeth procedure separately:

1. Import the libraries:

```
from statsmodels.api import OLS, add_constant
```

2. For the first step of the Fama-MacBeth regression, estimate the factor loadings:

```
factor_loadings = []
for portfolio in portfolio_df:
    reg_1 = OLS(
        endog=portfolio_df.loc[:, portfolio],
        exog=add_constant(factor_5_df)
    ).fit()
    factor_loadings.append(reg_1.params.drop("const"))
```

3. Store the factor loadings in a DataFrame:

```
factor_load_df = pd.DataFrame(
    factor_loadings,
    columns=factor_5_df.columns,
    index=portfolio_df.columns
)
factor_load_df.head()
```

Running the code generates the following table containing the factor loadings:

	Mkt-RF	SMB	HML	RMW	CMA
NoDur	0.786087	-0.215818	-0.083847	0.468129	0.338823
Durbl	1.548809	0.580849	-0.168357	0.294196	0.299400
Manuf	1.094951	0.291003	0.146831	0.086695	-0.010987
Enrgy	1.248025	0.487285	0.630805	0.243854	0.404512
Chems	0.885184	-0.089296	-0.018501	0.171997	0.210732

Figure 8.10: First step of the Fama-MacBeth regression—the estimated factor loadings

We can compare those numbers to the output of the full summary from the `linearmodels` library.

- For the second step of the Fama-MacBeth regression, estimate the risk premiums:

```
risk_premia = []
for period in portfolio_df.index:
    reg_2 = OLS(
        endog=portfolio_df.loc[period, factor_load_df.index],
        exog=factor_load_df
    ).fit()
    risk_premia.append(reg_2.params)
```

- Store the risk premiums in a DataFrame:

```
risk_premia_df = pd.DataFrame(
    risk_premia,
    index=portfolio_df.index,
    columns=factor_load_df.columns.tolist())
risk_premia_df.head()
```

Running the code generates the following table containing the risk premiums over time:

Date	Mkt-RF	SMB	HML	RMW	CMA
2010-01	-0.032631	0.051998	-0.023749	-0.039525	0.015071
2010-02	0.036662	0.020982	-0.014351	0.027181	-0.029331
2010-03	0.065954	-0.031731	-0.003074	-0.001531	-0.001160
2010-04	0.019455	0.048860	0.009688	0.040766	-0.014576
2010-05	-0.076882	0.024591	-0.021421	0.021403	-0.014296

Figure 8.11: Second step of the Fama-MacBeth regression—the estimated risk premiums over time

6. Calculate the average risk premiums:

```
risk_premia_df.mean()
```

Running the snippet returns:

Mkt-RF	0.012341
SMB	-0.006291
HML	-0.008927
RMW	-0.000908
CMA	-0.002484

The risk premiums calculated above match the ones obtained from the `linearmodels` library.

See also

- Documentation of the `linearmodels` library can be a good resource for learning about panel regression models (and not only that—it also contains utilities for instrumental variables models and so on) and their implementation in Python: <https://bashtage.github.io/linearmodels/index.html>
- Description of the 12 Industry Portfolios dataset: https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library/det_12_ind_port.html

Further reading about the Fama-MacBeth procedure:

- Fama, E. F., and MacBeth, J. D., “*Risk, return, and equilibrium: Empirical tests*,” *Journal of Political Economy*, 81, 3 (1973): 607-636
- Fama, E. F., “*Market efficiency, long-term returns, and behavioral finance*,” *Journal of Financial Economics*, 49, 3 (1998): 283-306

Summary

In this chapter, we have constructed some of the most popular factor models. We have started with the simplest one-factor model (the CAPM) and then explained how to approach more advanced three-, four-, and five-factor models. We have also described how we can use the Fama-MacBeth regression to estimate the factor models for multiple assets with appropriate cross-section (panel) data.

9

Modeling Volatility with GARCH Class Models

In *Chapter 6, Time Series Analysis and Forecasting*, we looked at various approaches to modeling time series. However, models such as ARIMA (Autoregressive Integrated Moving Average) cannot account for volatility that is not constant over time (heteroskedastic). We have already explained that some transformations (such as log or Box-Cox transformations) can be used to adjust for modest changes in volatility, but we would like to go a step further and model it.

In this chapter, we focus on **conditional heteroskedasticity**, which is a phenomenon caused when an increase in volatility is correlated with a further increase in volatility. An example might help to understand this concept. Imagine the price of an asset going down significantly due to some breaking news related to the company. Such a sudden price drop could trigger certain risk management tools of investment funds, which start selling the stocks as a result of the previous decrease in price. This could result in the price plummeting even further. Conditional heteroskedasticity was also clearly visible in the *Investigating stylized facts of asset returns* recipe, in which we showed that returns exhibit volatility clustering.

We would like to briefly explain the motivation for this chapter. Volatility is an incredibly important concept in finance. It is synonymous with risk and has many applications in quantitative finance. Firstly, it is used in options pricing, as the Black-Scholes model relies on the volatility of the underlying asset. Secondly, volatility has a significant impact on risk management, where it is used to calculate metrics such as the Value-at-Risk (VaR) of a portfolio, the Sharpe ratio, and many more. Thirdly, volatility is also present in trading. Normally, traders make decisions based on predictions of the assets' prices either rising or falling. However, we can also trade based on predicting whether there will be movement in any direction, that is, whether there will be volatility. **Volatility trading** is particularly appealing when certain world events (for example, pandemics) are driving markets to move erratically. An example of a product interesting to volatility traders might be the **Volatility Index (VIX)**, which is based on the movements of the S&P 500 index.

By the end of the chapter, we will have covered a selection of GARCH (Generalized Autoregressive Conditional Heteroskedasticity) models—both univariate and multivariate—which are some of the most popular ways of modeling and forecasting volatility. Knowing the basics, it is quite simple to implement more advanced models. We have already mentioned the importance of volatility in finance. By knowing how to model it, we can use such forecasts to replace the previously used naïve ones in many practical use cases in the fields of risk management or derivatives valuation.

In this chapter, we will cover the following recipes:

- Modeling stock returns' volatility with ARCH models
- Modeling stock returns' volatility with GARCH models
- Forecasting volatility using GARCH models
- Multivariate volatility forecasting with the CCC-GARCH model
- Forecasting the conditional covariance matrix using DCC-GARCH

Modeling stock returns' volatility with ARCH models

In this recipe, we approach the problem of modeling the conditional volatility of stock returns with the **Autoregressive Conditional Heteroskedasticity (ARCH)** model.

To put it simply, the ARCH model expresses the variance of the error term as a function of past errors. To be a bit more precise, it assumes that the variance of the errors follows an autoregressive model. The entire logic of the ARCH method can be represented by the following equations:

$$r_t = \mu + \epsilon_t$$

$$\epsilon_t = \sigma_t z_t$$

$$\sigma_t^2 = \omega + \sum_{i=1}^q \alpha_i \epsilon_{t-1}^2$$

The first equation represents the return series as a combination of the expected return μ and the unexpected return ϵ_t . ϵ_t has white noise properties—the conditional mean equal to zero and the time-varying conditional variance σ_t^2 .

Error terms are serially uncorrelated but do not need to be serially independent, as they can exhibit conditional heteroskedasticity.



ϵ_t is also known as the mean-corrected return, error term, innovations, or—most commonly—residuals.

In general, ARCH (and GARCH) models should only be fitted to the residuals of some other model applied to the original time series. When estimating volatility models, we can assume different specifications of the mean process, for example:

- A zero-mean process—this implies that the returns are only described by the residuals, for example, $r_t = \epsilon_t$
- A constant mean process ($r_t = \mu + \epsilon_t$)
- Mean estimated using linear models such as AR, ARMA, ARIMA, or the more recent heterogeneous autoregressive (HAR) process

In the second equation, we represent the error series in terms of a stochastic component $z_t \sim N(0,1)$ and a conditional standard deviation σ_t , which governs the typical size of the residuals. The stochastic component can also be interpreted as standardized residuals.

The third equation presents the ARCH formula, where $\omega > 0$ and $\alpha_i \geq 0$. Some important points about the ARCH model include:

- The ARCH model explicitly recognizes the difference between the unconditional and the conditional variance of the time series.
- It models the conditional variance as a function of past residuals (errors) from a mean process.
- It assumes the unconditional variance to be constant over time.
- The ARCH model can be estimated using the ordinary least squares (OLS) method.
- We must specify the number of prior residuals (q) in the model—similarly to the AR model.
- The residuals should look like observations of a discrete white noise—zero-mean and stationary (no trends or seasonal effects, that is, no evident serial correlation).



In the original ARCH notation, as well as in the `arch` library in Python, the lag hyperparameter is denoted with p . However, we use q as the corresponding symbol, in line with the GARCH notation introduced in the next recipe.

The biggest strength of the ARCH model is that the volatility estimates it produces exhibit excess kurtosis (fat tails as compared to Normal distribution), which is in line with the empirical observations about stock returns. Naturally, there are also weaknesses. The first one is that the model assumes the same effects of positive and negative volatility shocks, which is simply not the case. Secondly, it does not explain variations in volatility. That is why the model is likely to over-forecast volatility, as it is slow to respond to large, isolated shocks in the returns series.

In this recipe, we fit the ARCH(1) model to Google's daily stock returns from the years 2015 to 2021.

How to do it...

Execute the following steps to fit the ARCH(1) model:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
from arch import arch_model
```

2. Specify the risky asset and the time horizon:

```
RISKY_ASSET = "GOOG"  
START_DATE = "2015-01-01"  
END_DATE = "2021-12-31"
```

3. Download data from Yahoo Finance:

```
df = yf.download(RISKY_ASSET,  
                 start=START_DATE,  
                 end=END_DATE,  
                 adjusted=True)
```

4. Calculate the daily returns:

```
returns = 100 * df["Adj Close"].pct_change().dropna()  
returns.name = "asset_returns"  
returns.plot(  
    title=f"{RISKY_ASSET} returns: {START_DATE} - {END_DATE}"  
)
```

Running the code generates the following plot:

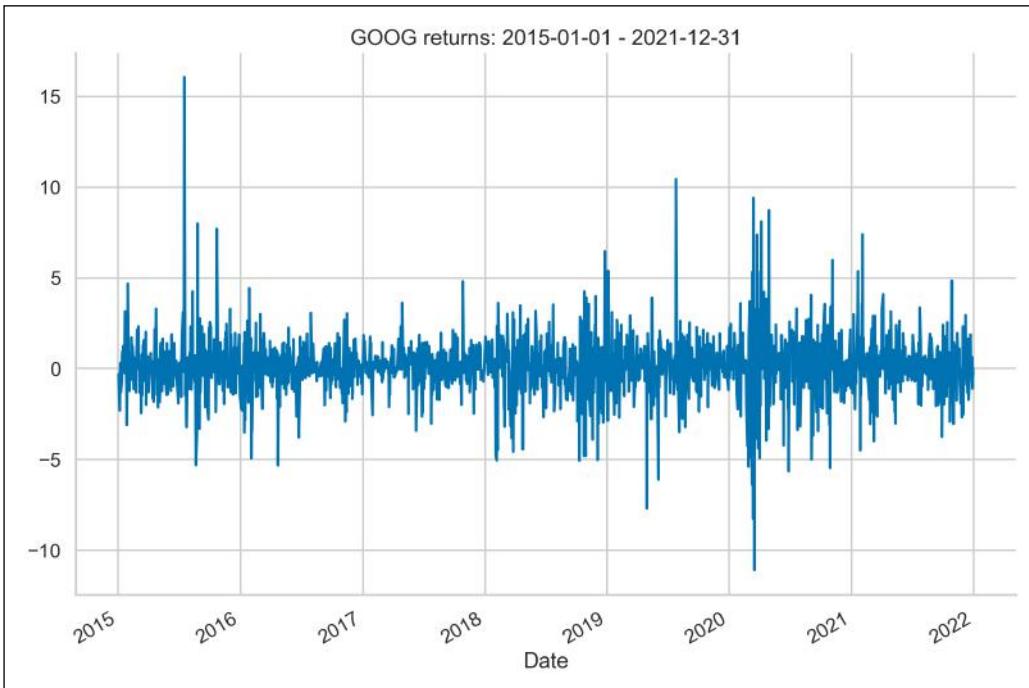


Figure 9.1: Google's simple returns from the years 2015 to 2021

In the plot, we can observe a few sudden spikes and clear examples of volatility clustering.

5. Specify the ARCH model:

```
model = arch_model(returns, mean="Zero", vol="ARCH", p=1, q=0)
```

6. Estimate the model and print the summary:

```
fitted_model = model.fit(disp="off")
print(fitted_model.summary())
```

Running the code returns the following summary:

```
Zero Mean - ARCH Model Results
=====
Dep. Variable: asset_returns    R-squared:      0.000
Mean Model:      Zero Mean     Adj. R-squared:   .001
Vol Model:       ARCH          Log-Likelihood: -3302.93
Distribution:    Normal         AIC:            6609.85
Method:          Maximum        BIC:            6620.80
                           Likelihood
                                         No. Observations: 1762
Date:             Wed, Jun 08 2022 Df Residuals:    1762
Time:                  22:25:16   Df Model:       0
                           Volatility Model
=====
              coef      std err      t      P>|t|  95.0% Conf. Int.
-----
omega      1.8625    0.166    11.248  2.359e-29 [ 1.538, 2.187]
alpha[1]    0.3788    0.112     3.374  7.421e-04 [ 0.159, 0.599]
=====
```

7. Plot the residuals and the conditional volatility:

```
fitted_model.plot(annualize="D")
```

Running the code results in the following plots:

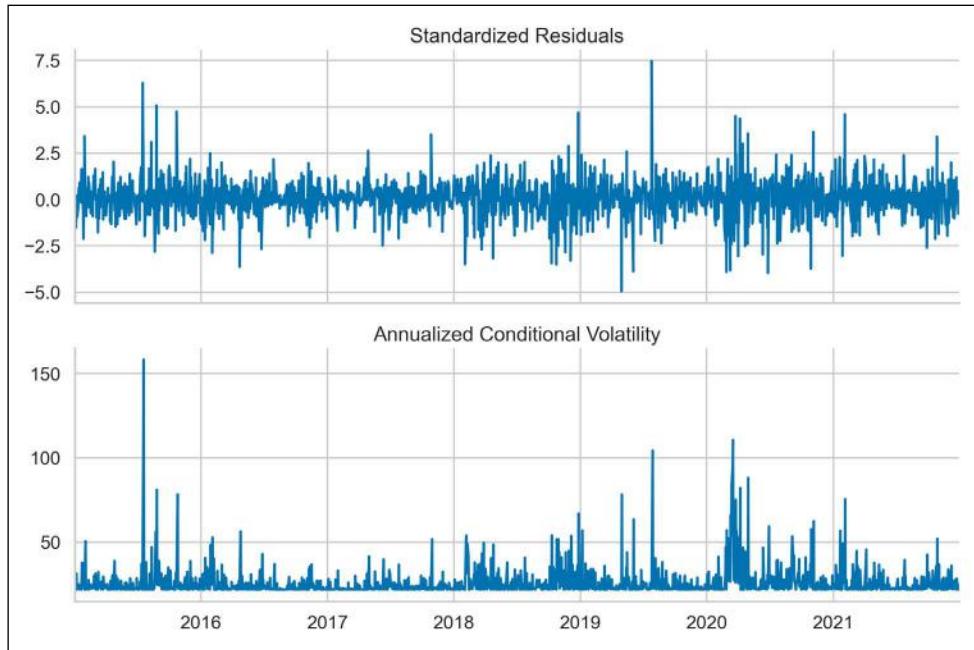


Figure 9.2: Standardized residuals and the annualized conditional volatility of the fitted ARCH model

We can observe some standardized residuals that are large (in magnitude) and correspond to highly volatile periods.

How it works...

In *Steps 2 to 4*, we downloaded Google's daily stock prices and calculated simple returns. When working with ARCH/GARCH models, convergence warnings are likely to occur in the case of very small numbers. This is caused by instabilities in the underlying optimization algorithms of the `scipy` library. To overcome this issue, we multiplied the returns by 100 to express them as percentages.

In *Step 5*, we defined the ARCH(1) model. For the mean model, we selected the zero-mean approach, which is suitable for many liquid financial assets. Another viable choice here could be a constant mean. We can use those approaches as opposed to, for example, ARMA models because the serial dependence of the return series might be very limited.

In *Step 6*, we fitted the model using the `fit` method. Additionally, we passed `disp="off"` to the `fit` method to suppress output from the optimization steps. To fit the model using the `arch` library, we had to take similar steps to the familiar `scikit-learn` approach: we first defined the model and then fitted it to the data. One difference would be the fact that with `arch`, we had to provide the data object while creating the instance of the model, instead of passing it to the `fit` method as we would have done in `scikit-learn`. Then, we printed the model's summary by using the `summary` method.

In Step 7, we also inspected the standardized residuals and the conditional volatility series by plotting them. The standardized residuals were computed by dividing the residuals by the conditional volatility. We passed `annualize="D"` to the `plot` method in order to annualize the conditional volatility series from daily data.

There's more...

A few more noteworthy points about ARCH models:

- Selecting the zero-mean process is useful when working on residuals from a separately estimated model.
- To detect ARCH effects, we can look at the correlogram of the squared residuals from a certain model (such as the ARIMA model). We need to make sure that the mean of these residuals is equal to zero. We can use the Partial Autocorrelation Function (PACF) plot to infer the value of q , similarly to the approach used in the case of the AR model (please refer to the *Modeling time series with ARIMA class models* recipe for more details).
- To test the validity of the model, we can inspect whether the standardized residuals and squared standardized residuals exhibit no serial autocorrelation (for example, using the Ljung-Box or Box-Pierce test with the `acorr_ljungbox` function from `statsmodels`). Alternatively, we can employ the **Lagrange Multiplier test** (the LM test, also known as Engle's Test for Autoregressive Conditional Heteroscedasticity) to make sure that the model captures all ARCH effects. To do so, we can use the `het_arch` function from `statsmodels`.

In the following snippet, we test the residuals of the ARCH model with the LM test:

```
from statsmodels.stats.diagnostic import het_arch
het_arch(fitted_model.resid)
```

Running the code returns the following tuple:

```
(98.10927835448403,
 1.3015895084238874e-16,
 10.327662606705564,
 4.2124269229123006e-17)
```

The first two values in the tuple are the LM test statistic and its corresponding p-value. The latter two are the f-statistic for the F test (an alternative approach to testing for ARCH effects) and its corresponding p-value. We can see that both p-values are below the customary significance level of 0.05, which leads us to reject the null hypothesis stating that the residuals are homoskedastic. This means that the ARCH(1) model fails to capture all ARCH effects in the residuals.



The documentation of the `het_arch` function suggests that if the residuals are coming from a regression model, we should correct for the number of estimated parameters in that model. For example, if the residuals were coming from an ARMA(2, 1) model, we should pass an additional argument to the `het_arch` function, `ddof = 3`, where `ddof` stands for the degrees of freedom.

See also

Additional resources are available here:

- Engle, R. F. 1982., “Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation,” *Econometrica*, 50(4): 987-1007

Modeling stock returns' volatility with GARCH models

In this recipe, we present how to work with an extension of the ARCH model, namely the **Generalized Autoregressive Conditional Heteroskedasticity (GARCH)** model. GARCH can be considered an ARMA model applied to the variance of a time series—the AR component was already expressed in the ARCH model, while GARCH additionally adds the moving average part.

The equation of the GARCH model can be presented as:

$$r_t = \mu + \epsilon_t$$

$$\epsilon_t = \sigma_t z_t$$

$$\sigma_t^2 = \omega + \sum_{i=1}^q \alpha_i \epsilon_{t-1}^2 + \sum_{i=1}^p \beta_i \sigma_{t-1}^2$$

While the interpretation is very similar to the ARCH model presented in the previous recipe, the difference lies in the last equation, where we can observe an additional component. Parameters are constrained to meet the following: $\omega > 0$, $\alpha_i \geq 0$, and $\beta_i \geq 0$.



In the GARCH model, there are additional constraints on coefficients. For example, in the case of a GARCH(1,1) model, $\alpha_i + \beta_i$ must be less than 1. Otherwise, the model is unstable.

The two hyperparameters of the GARCH model can be described as:

- p : The number of lag variances
- q : The number of lag residual errors from a mean process



A GARCH(0, q) model is equivalent to an ARCH(q) model.

One way of inferring the lag orders for ARCH/GARCH models is to use the squared residuals from a model used to predict the mean of the original time series. As the residuals are centered around zero, their squares correspond to their variance. We can inspect the ACF/PACF plots of the squared residuals in order to identify patterns in the autocorrelation of the series' variance (similarly to what we have done to identify the orders of an ARMA/ARIMA model).

In general, the GARCH model shares the strengths and weaknesses of the ARCH model, with the difference that it better captures the effects of past shocks. Please see the *There's more...* section to learn about some extensions of the GARCH model that account for the original model's shortcomings.

In this recipe, we apply the GARCH(1,1) model to the same data as in the previous recipe, in order to clearly highlight the difference between the two modeling approaches.

How to do it...

Execute the following steps to estimate the GARCH(1,1) model in Python:

1. Specify the GARCH model:

```
model = arch_model(returns, mean="Zero", vol="GARCH", p=1, q=1)
```

2. Estimate the model and print the summary:

```
fitted_model = model.fit(disp="off")
print(fitted_model.summary())
```

Running the code returns the following summary:

```
Zero Mean - GARCH Model Results
=====
Dep. Variable: asset_returns R-squared: 0.000
Mean Model: Zero Mean Adj. R-squared: 0.001
Vol Model: GARCH Log-Likelihood: -3246.71
Distribution: Normal AIC: 6499.42
Method: Maximum BIC: 6515.84
Likelihood
No. Observations: 1762
Date: Wed, Jun 08 2022 Df Residuals: 1762
Time: 22:37:27 Df Model: 0
Volatility Model
=====
      coef    std err      t    P>|t|    95.0% Conf. Int.
-----
omega    0.2864    0.186    1.539    0.124    [-7.844e-02, 0.651]
alpha[1]  0.1697  9.007e-02   1.884  5.962e-02  [-6.879e-03, 0.346]
beta[1]   0.7346    0.128    5.757  8.538e-09   [ 0.485, 0.985]
=====
```

According to *Market Risk Analysis*, the usual range of values of the parameters in a stable market would be $0.05 < \alpha < 0.01$ and $0.85 < \beta < 0.98$. However, we should keep in mind that while these ranges will most likely not strictly apply, they already give us some idea of what kinds of values we should be expecting.

We can see that, compared to the ARCH model, the log-likelihood increased, which means that the GARCH model fits the data better. However, we should be cautious when drawing such conclusions. The log-likelihood will most likely increase every time we add more predictors (as we have done with GARCH). In case the number of predictors changes, we should run a likelihood-ratio test in order to compare the goodness-of-fit criteria of two nested regression models.

3. Plot the residuals and the conditional volatility:

```
fitted_model.plot(annualize="D")
```

In the plots below, we can observe the effect of including the extra component (lagged conditional volatility) into the model specification:

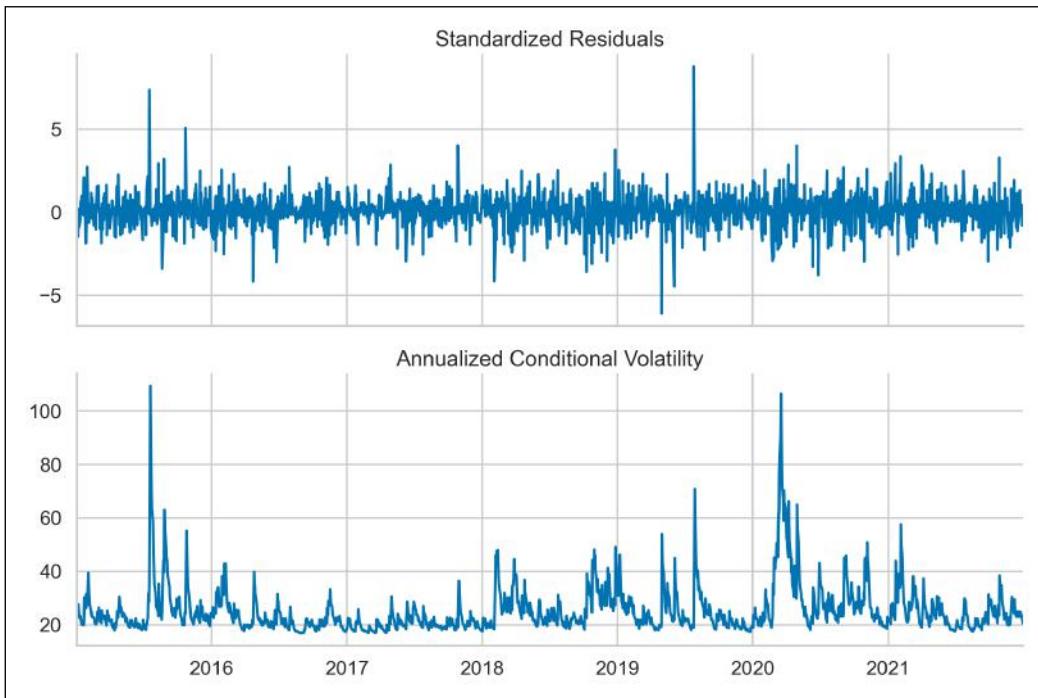


Figure 9.3: Standardized residuals and the annualized conditional volatility of the fitted GARCH model

When using ARCH, the conditional volatility series exhibits many spikes, and then immediately returns to a low level. In the case of GARCH, as the model also includes lagged conditional volatility, it takes more time to return to the level observed before the spike.

How it works...

In this recipe, we used the same data as in the previous one to compare the results of the ARCH and GARCH models. For more information on downloading data, please refer to *Steps 1 to 4* in the *Modeling stock returns' volatility with ARCH models* recipe.

Due to the convenience of the `arch` library, it was very easy to adjust the code used previously to fit the ARCH model. To estimate the GARCH model, we had to specify the type of volatility model we wanted to use and set an additional argument: `q=1`.

For comparison's sake, we left the mean process as a zero-mean process.

There's more...

In this chapter, we have already used two models to explain and potentially forecast the conditional volatility of a time series. However, there are numerous extensions of the GARCH model, as well as different configurations with which we can experiment in order to find the best-fitting model.

In the GARCH framework, aside from the hyperparameters (such as p and q , in the case of the vanilla GARCH model), we can modify the models described next.

Conditional mean model

As explained before, we apply the GARCH class models to residuals obtained after fitting another model to the series. Some popular choices for the mean model are:

- Zero-mean
- Constant mean
- Any variant of the ARIMA model (including potential seasonality adjustment, as well as external regressors)—some popular choices in the literature are ARMA or even AR models
- Regression models



We should be aware of one thing when modeling the conditional mean. For example, we may first fit an ARMA model to our time series and then fit a GARCH model to the residuals of the first model. However, this is not the preferred way. That is because, in general, the ARMA estimates will be inconsistent (or consistent but inefficient, in the case when there are only AR terms and no MA terms), which will also impact the following GARCH estimates. The inconsistency arises because the first model (ARMA/ARIMA) assumes conditional homoskedasticity, while we are explicitly modeling conditional heteroskedasticity with the GARCH model in the second step. That is why the preferred way is to estimate both models simultaneously, for example, using the `arch` library (or the `rugarch` package for R).

Conditional volatility model

There are numerous extensions to the GARCH framework. Some popular models include:

- **GJR-GARCH:** A variant of the GARCH model that takes into account the asymmetry of the returns (negative returns tend to have a stronger impact on volatility than positive ones)
- **EGARCH:** Exponential GARCH
- **TGARCH:** Threshold GARCH
- **FIGARCH:** Fractionally integrated GARCH, used with non-stationary data

- GARCH-MIDAS: In this class of models, volatility is decomposed into a short-term GARCH component and a long-term component driven by an additional explanatory variable
- Multivariate GARCH models, such as CCC-/DCC-GARCH

The first three models use slightly different approaches to introduce asymmetry into the conditional volatility specification. This is in line with the belief that negative shocks have a stronger impact on volatility than positive shocks.

Distribution of errors

In the *Investigating stylized facts of asset returns* recipe, we saw that the distribution of returns is not Normal (skewed, with heavy tails). That is why distributions other than Gaussian might be more fitting for errors in the GARCH model.

Some possible choices are:

- Student's t-distribution
- Skew-t distribution (Hansen, 1994)
- Generalized Error Distribution (GED)
- Skewed Generalized Error Distribution (SGED)



The `arch` library not only provides most of the models and distributions mentioned above, but it also allows for the use of your own volatility models/distributions of errors (as long as they fit into a predefined format). For more information on this, please refer to the excellent documentation.

See also

Additional resources are available here:

- Alexander, C. 2008. *Market Risk Analysis, Practical Financial Econometrics* (Vol. 2). John Wiley & Sons.
- Bollerslev, T., 1986. “Generalized Autoregressive Conditional Heteroskedasticity. *Journal of Econometrics*, 31, (3): 307–327. : [https://doi.org/10.1016/0304-4076\(86\)90063-1](https://doi.org/10.1016/0304-4076(86)90063-1)
- Glosten, L. R., Jagannathan, R., and Runkle, D. E., 1993. “On the relation between the expected value and the volatility of the nominal excess return on stocks,” *The Journal of Finance*, 48 (5): 1779–1801: <https://doi.org/10.1111/j.1540-6261.1993.tb05128.x>
- Hansen, B. E., 1994. “Autoregressive conditional density estimation,” *International Economic Review*, 35(3): 705–730: <https://doi.org/10.2307/2527081>
- Documentation of the `arch` library—<https://arch.readthedocs.io/en/latest/index.html>

Forecasting volatility using GARCH models

In the previous recipes, we have seen how to fit ARCH/GARCH models to a return series. However, the most interesting/relevant case of using ARCH class models would be to forecast the future values of the volatility.

There are three approaches to forecasting volatility using GARCH class models:

- Analytical – due to the inherent structure of ARCH class models, analytical forecasts are always available for the one step-ahead forecast. Multi-step analytical forecasts can be obtained using a forward recursion; however, that is only possible for models that are linear in the square of the residuals (such as GARCH or Heterogeneous ARCH).
- Simulation—simulation-based forecasts use the structure of an ARCH class model to forward simulate possible volatility paths using the assumed distribution of residuals. In other words, they use random number generators (assuming specific distributions) to draw the standardized residuals. This approach creates x possible volatility paths and then produces the average as the final forecast. Simulation-based forecasts are always available for any horizon. As the number of simulations increases toward infinity, the simulation-based forecasts will converge to the analytical forecasts.
- Bootstrap (also known as the Filtered Historical Simulation)—those forecasts are very similar to the simulation-based forecasts with the difference that they generate (to be precise, draw with replacement) the standardized residuals using the actual input data and the estimated parameters. This approach requires a minimal amount of in-sample data to use prior to producing the forecasts.



Due to the specification of ARCH class models, the first out-of-sample forecast will always be fixed, regardless of which approach we use.

In this recipe, we fit a GARCH(1,1) model with Student's t distributed residuals to Microsoft's stock returns from the years 2015 to 2020. Then, we create 3-step ahead forecasts for each day of 2021.

How to do it...

Execute the following steps to create 3-step ahead volatility forecasts using a GARCH model:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
from datetime import datetime
from arch import arch_model
```

2. Download data from Yahoo Finance and calculate simple returns:

```
df = yf.download("MSFT",
                 start="2015-01-01",
                 end="2021-12-31",
                 adjusted=True)

returns = 100 * df["Adj Close"].pct_change().dropna()
returns.name = "asset_returns"
```

3. Specify the GARCH model:

```
model = arch_model(returns, mean="Zero", vol="GARCH", dist="t",
                    p=1, q=1)
```

4. Define the split date and fit the model:

```
SPLIT_DATE = datetime(2021, 1, 1)
fitted_model = model.fit(last_obs=SPLIT_DATE, disp="off")
```

5. Create and inspect the analytical forecasts:

```
forecasts_analytical = fitted_model.forecast(horizon=3,
                                              start=SPLIT_DATE,
                                              reindex=False)

forecasts_analytical.variance.plot(
    title="Analytical forecasts for different horizons"
)
```

Running the snippet generates the following plot:

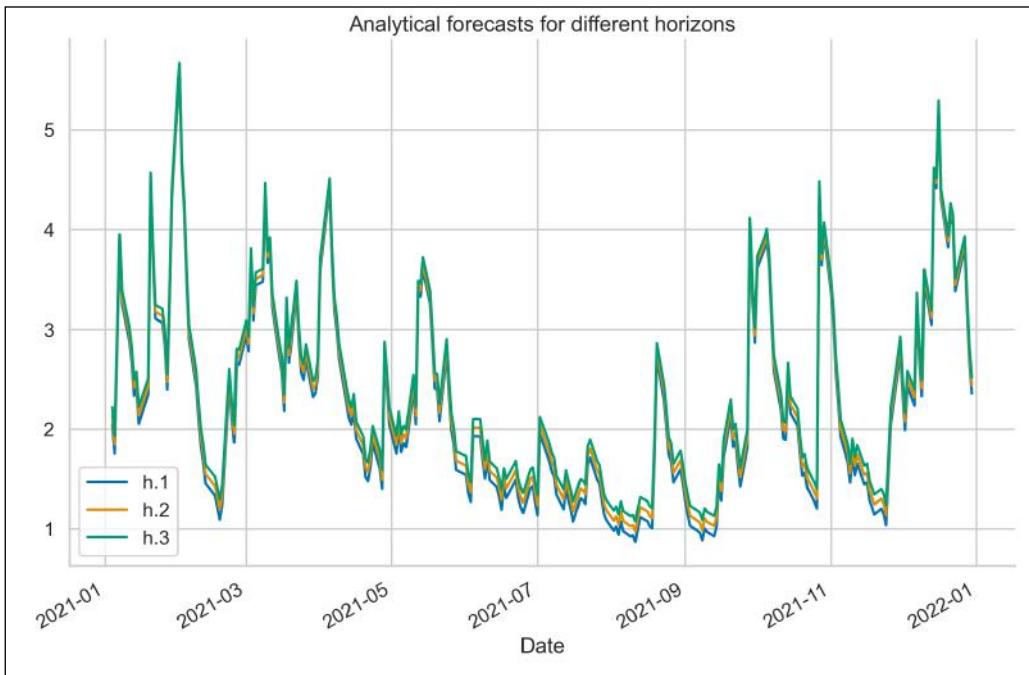


Figure 9.4: Analytical forecasts for horizons 1, 2, and 3

Using the snippet below, we can inspect the generated forecasts.

```
forecasts_analytical.variance
```

Date	h.1	h.2	h.3
2021-01-04	2.050335	2.136144	2.220612
2021-01-05	1.755680	1.846097	1.935101
2021-01-06	2.772100	2.846622	2.919980
2021-01-07	3.839409	3.897241	3.954169
2021-01-08	3.250745	3.317782	3.383772
...
2021-12-23	3.385120	3.450056	3.513977
2021-12-27	3.821128	3.879246	3.936455
2021-12-28	3.189848	3.257838	3.324764
2021-12-29	2.671069	2.747172	2.822084
2021-12-30	2.359531	2.440505	2.520213

Figure 9.5: Table presenting the analytical forecasts for horizons 1, 2, and 3

Each column contains the h -step ahead forecasts generated on the date indicated by the index. When the forecasts are created, the date from the Date column corresponds to the last data point used to generate the forecasts. For example, the columns with the date 2021-01-08 contain the forecasts for January 9, 10, and 11. Those forecasts were created using data up to and including January 8.

6. Create and inspect the simulation forecasts:

```
forecasts_simulation = fitted_model.forecast(
    horizon=3,
    start=SPLIT_DATE,
    method="simulation",
    reindex=False
)

forecasts_simulation.variance.plot(
    title="Simulation forecasts for different horizons"
)
```

Running the snippet generates the following plot:

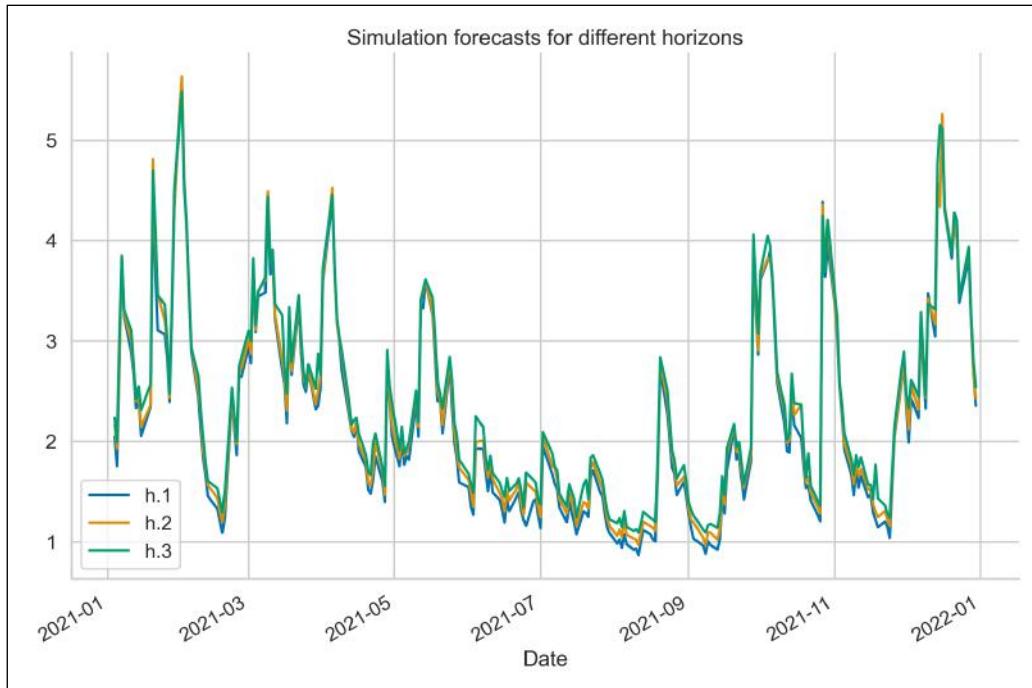


Figure 9.6: Simulation-based forecasts for horizons 1, 2, and 3

7. Create and inspect the bootstrap forecasts:

```
forecasts_bootstrap = fitted_model.forecast(horizon=3,
                                             start=SPLIT_DATE,
                                             method="bootstrap",
                                             reindex=False)

forecasts_bootstrap.variance.plot(
    title="Bootstrap forecasts for different horizons"
)
```

Running the snippet generates the following plot:

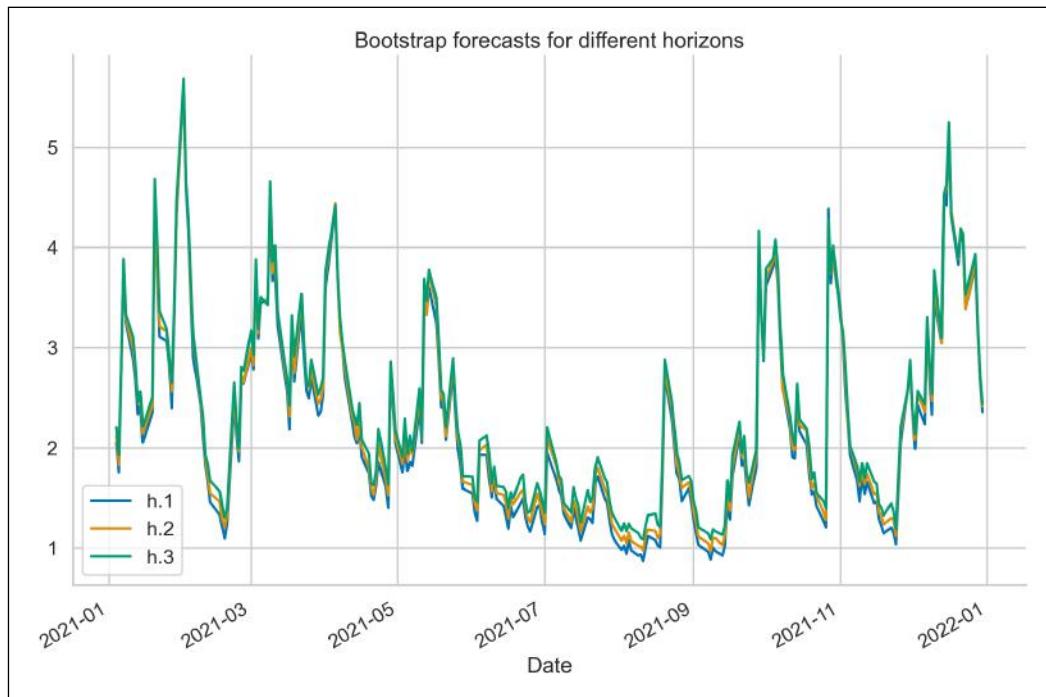


Figure 9.7: Bootstrap-based forecasts for horizons 1, 2, and 3

Inspecting the three plots leads to the conclusion that the shape of the volatility forecasts from the three different methods is very similar.

How it works...

In the first two steps, we imported the required libraries and downloaded Microsoft's stock prices from the years 2015 to 2021. We calculated the simple returns and multiplied the values by 100 to avoid potential convergence issues during optimization.


```
        reindex=False)
    .residual_variance["2020"]
    .apply(np.sqrt)
)

vol_bootstrap = (
    fitted_model.forecast(horizon=FCST_HORIZON,
                          start=datetime(2020, 1, 1),
                          method="bootstrap",
                          reindex=False)
    .residual_variance["2020"]
    .apply(np.sqrt)
)
```

While creating the forecasts, we changed the horizon and the start date. We recovered the residual variance from the fitted models, filtered for the forecasts made in 2020, and then took the square root to convert the variance into volatility.

3. Get the conditional volatility for 2020:

```
vol = fitted_model.conditional_volatility["2020"]
```

4. Create the hedgehog plot:

```
ax = vol.plot(
    title="Comparison of analytical vs bootstrap volatility forecasts",
    alpha=0.5
)
ind = vol.index
for i in range(0, 240, 10):
    vol_a = vol_analytic.iloc[i]
    vol_b = vol_bootstrap.iloc[i]
    start_loc = ind.get_loc(vol_a.name)
    new_ind = ind[(start_loc+1):(start_loc+FCST_HORIZON+1)]
    vol_a.index = new_ind
    vol_b.index = new_ind
    ax.plot(vol_a, color="r")
    ax.plot(vol_b, color="g")

labels = ["Volatility", "Analytical Forecast",
          "Bootstrap Forecast"]
legend = ax.legend(labels)
```

Running the snippet generates the following plot:

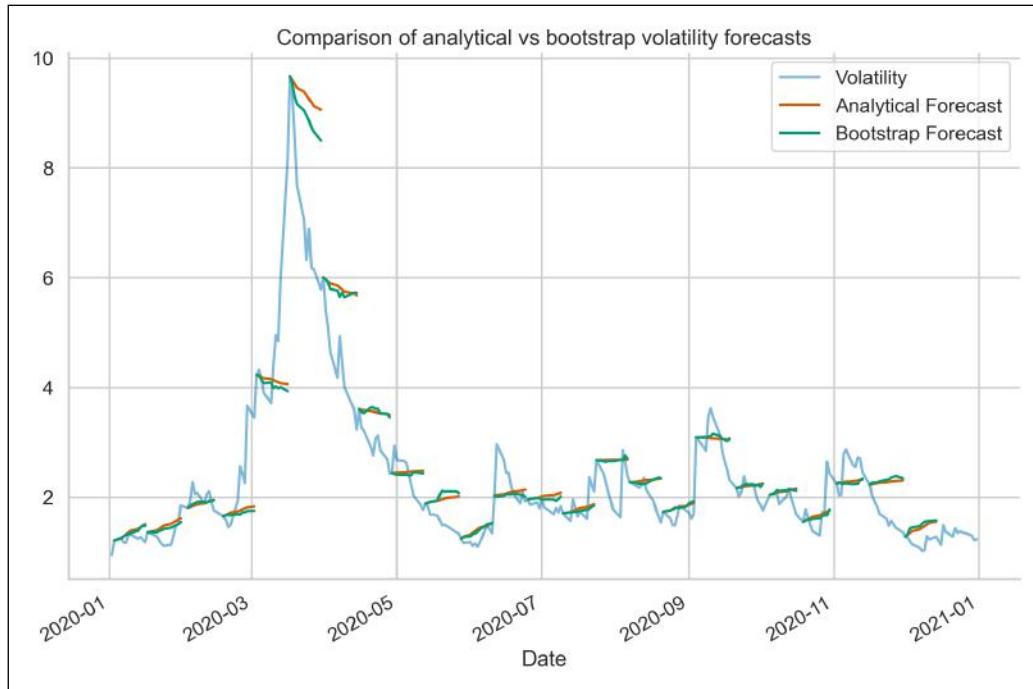


Figure 9.8: Comparison of analytical and bootstrap-based approaches to volatility forecasting

A hedgehog plot is a useful kind of visualization for showing the differences between the two forecasting approaches over a longer period of time. In this case, we plotted the 10-step ahead forecasts every 10 days.

What is interesting to note is the peak in volatility that occurred in March 2020. We can see that close to the peak, the GARCH model is predicting a decrease in volatility over the next few days. To get a better understanding of how that forecast was created, we can refer to the underlying data. By inspecting the DataFrames containing the observed volatility and the forecasts, we can state that the peak happened on March 17, while the plotted forecast was created using data up until March 16.



When inspecting a single volatility model at a time, it might be easier to use the `hedgehog_plot` method of the fitted `arch_model` to create a similar plot.

Multivariate volatility forecasting with the CCC-GARCH model

In this chapter, we have already considered multiple univariate conditional volatility models. That is why, in this recipe, we move to the multivariate setting. As a starting point, we consider Bollerslev's **Constant Conditional Correlation GARCH** (CCC-GARCH) model. The idea behind it is quite simple. The model consists of N univariate GARCH models, related to each other via a constant conditional correlation matrix \mathbf{R} .

Like before, we start with the model's specification:

$$\mathbf{r}_t = \boldsymbol{\mu} + \boldsymbol{\epsilon}_t$$

$$\boldsymbol{\epsilon}_t \sim N(0, \boldsymbol{\Sigma}_t)$$

$$\boldsymbol{\Sigma}_t = \mathbf{D}_t \mathbf{R} \mathbf{D}_t$$

In the first equation, we represent the return series. The key difference between this representation and the one presented in previous recipes is the fact that this time, we are considering multivariate returns. That is why \mathbf{r}_t is actually a vector of returns $\mathbf{r}_t = (r_{1t}, \dots, r_{nt})$. The mean and error terms are represented analogically. To highlight this, we use bold font when considering vectors or matrices.

The second equation shows that the error terms come from a Multivariate Normal distribution with zero means and a conditional covariance matrix $\boldsymbol{\Sigma}_t$ (of size $N \times N$).

The elements of the conditional covariance matrix are defined as:

- Diagonal: $\sigma_{ii,t}^2 = \omega_{ii} + \sum_{i=1}^q \alpha_{ii} \epsilon_{i,t-i}^2 + \sum_{i=1}^p \beta_{ii} \sigma_{i,t-i}^2 \quad \text{for } i = 1, \dots, N$
- Off-diagonal: $\sigma_{ij,t}^2 = \rho_{ij} \sigma_{ii,t} \sigma_{jj,t} \quad \text{for } i \neq j$

The third equation presents the decomposition of the conditional covariance matrix. \mathbf{D}_t represents a matrix containing the conditional standard deviations on the diagonal, and \mathbf{R} is a correlation matrix.

The key ideas of the model are as follows:

- The model avoids the problem of guaranteeing positive definiteness of $\boldsymbol{\Sigma}_t$ by splitting it into variances and correlations.
- The conditional correlations between error terms are constant over time.
- Individual conditional variances follow a univariate GARCH(1,1) model.

In this recipe, we estimate the CCC-GARCH model on a series of stock returns for three US tech companies. For more details about the estimation of the CCC-GARCH model, please refer to the *How it works...* section.

How to do it...

Execute the following steps to estimate the CCC-GARCH model in Python:

1. Import the libraries:

```
import pandas as pd
import numpy as np
import yfinance as yf
from arch import arch_model
```

2. Specify the risky assets and the time horizon:

```
RISKY_ASSETS = ["GOOG", "MSFT", "AAPL"]
START_DATE = "2015-01-01"
END_DATE = "2021-12-31"
```

3. Download data from Yahoo Finance:

```
df = yf.download(RISKY_ASSETS,
                  start=START_DATE,
                  end=END_DATE,
                  adjusted=True)
```

4. Calculate the daily returns:

```
returns = 100 * df["Adj Close"].pct_change().dropna()
returns.plot(
    subplots=True,
    title=f"Stock returns: {START_DATE} - {END_DATE}"
)
```

Running the snippet generates the following plot:



Figure 9.9: Simple returns of Apple, Google, and Microsoft

5. Define lists for storing objects:

```
coeffs = []
cond_vol = []
std_resids = []
models = []
```

6. Estimate the univariate GARCH models:

```

for asset in returns.columns:
    model = arch_model(returns[asset], mean="Constant",
                       vol="GARCH", p=1, q=1)
    model = model.fit(update_freq=0, disp="off");
    coeffs.append(model.params)
    cond_vol.append(model.conditional_volatility)
    std_resids.append(model.std_resid)
    models.append(model)

```

7. Store the results in DataFrames:

```

coeffs_df = pd.DataFrame(coeffs, index=returns.columns)
cond_vol_df = (
    pd.DataFrame(cond_vol)
    .transpose()
    .set_axis(returns.columns,
              axis="columns")
)
std_resids_df = (
    pd.DataFrame(std_resids)
    .transpose()
    .set_axis(returns.columns
              axis="columns")
)

```

The following table contains the estimated coefficients for each return series:

	mu	omega	alpha[1]	beta[1]
AAPL	0.189287	0.176098	0.134563	0.811815
GOOG	0.124954	0.304484	0.183056	0.716245
MSFT	0.149370	0.268674	0.213800	0.699576

Figure 9.10: Coefficients of the estimated univariate GARCH models

8. Calculate the constant conditional correlation matrix (R):

```

R = (
    std_resids_df
    .transpose()
    .dot(std_resids_df)
    .div(len(std_resids_df))
)

```

9. Calculate the one-step ahead forecast of the conditional covariance matrix:

```
# define objects
diag = []
D = np.zeros((len(RISKY_ASSETS), len(RISKY_ASSETS)))

# populate the list with conditional variances
for model in models:
    diag.append(model.forecast(horizon=1).variance.iloc[-1, 0])
# take the square root to obtain volatility from variance
diag = np.sqrt(diag)
# fill the diagonal of D with values from diag
np.fill_diagonal(D, diag)

# calculate the conditional covariance matrix
H = np.matmul(np.matmul(D, R.values), D)
```

The calculated one-step ahead forecast looks as follows:

```
array([[2.39962391, 1.00627878, 1.19839517],
       [1.00627878, 1.51608369, 1.12048865],
       [1.19839517, 1.12048865, 1.87399738]])
```

We can compare this matrix to the one obtained using a more complex DCC-GARCH model, which we cover in the next recipe.

How it works...

In *Steps 2* and *Step 3*, we downloaded the daily stock prices of Google, Microsoft, and Apple. Then, we calculated simple returns and multiplied them by 100 to avoid encountering convergence errors.

In *Step 5*, we defined empty lists for storing elements required at later stages: GARCH coefficients, conditional volatilities, standardized residuals, and the models themselves (used for forecasting).

In *Step 6*, we iterated over the columns of the DataFrame containing the stock returns and fitted a univariate GARCH model to each of the series. We stored the results in the predefined lists. Then, we wrangled the data in order to have objects such as residuals in DataFrames, to make working with them easier.

In *Step 8*, we calculated the constant conditional correlation matrix (\mathbf{R}) as the unconditional correlation matrix of \mathbf{z}_t :

$$\mathbf{R} = \frac{1}{T} \sum_{t=1}^T \mathbf{z}_t \mathbf{z}_t'$$

Here, \mathbf{z}_t stands for time t standardized residuals from the univariate GARCH models.

In the last step, we obtained one-step ahead forecasts of the conditional covariance matrix H_{t+1} . To do so, we did the following:

- We created a matrix D_{t+1} of zeros, using `np.zeros`.
- We stored the one-step ahead forecasts of conditional variances from univariate GARCH models in a list called `diag`.
- Using `np.fill_diagonal`, we placed the elements of the list called `diag` on the diagonal of the matrix D_{t+1} .
- Following equation 3 from the introduction, we obtained the one-step ahead forecast using matrix multiplication (`np.matmul`).

See also

Additional resources are available here:

- Bollerslev, T.1990. “Modeling the Coherence in Short-Run Nominal Exchange Rates: A Multivariate Generalized ARCH Approach,” *Review of Economics and Statistics*, 72(3): 498–505: <https://doi.org/10.2307/2109358>

Forecasting the conditional covariance matrix using DCC-GARCH

In this recipe, we cover an extension of the CCC-GARCH model: Engle’s **Dynamic Conditional Correlation GARCH (DCC-GARCH)** model. The main difference between the two is that in the latter, the conditional correlation matrix is not constant over time—we work with R_t instead of R .

There are some nuances in terms of estimation, but the outline is similar to the CCC-GARCH model:

- Estimate the univariate GARCH models for conditional volatility
- Estimate the DCC model for conditional correlations

In the second step of estimating the DCC model, we use a new matrix Q_t , representing a proxy correlation process.

$$R_t = \text{diag}(Q_t)^{-1/2} Q_t \text{diag}(Q_t)^{-1/2}$$

$$Q_t = (1 - \gamma - \delta) \bar{Q} + \gamma z_{t-1} z'_{t-1} + \delta Q_{t-1}$$

$$\bar{Q} = \frac{1}{T} \sum_{t=1}^T z_t z'_{t-1}$$

The first equation describes the relationship between the conditional correlation matrix R_t and the proxy process Q_t . The second equation represents the dynamics of the proxy process. The last equation shows the definition of \bar{Q} , which is defined as the unconditional correlation matrix of standardized residuals from the univariate GARCH models.

This representation of the DCC model uses an approach called **correlation targeting**. It means that we are effectively reducing the number of parameters we need to estimate to two: γ and δ . This is similar to volatility targeting in the case of univariate GARCH models, further described in the *There's more...* section.

At the time of writing, there is no Python library that we can use to estimate DCC-GARCH models. One solution would be to write such a library from scratch. Another, more time-efficient solution would be to use a well-established R package for that task. That is why in this recipe, we also introduce how to efficiently make Python and R work together in one Jupyter notebook (this can also be done in a normal .py script). The rpy2 library is an interface between both languages. It enables us to not only run both R and Python in the same notebook but also to transfer objects between the two environments.

In this recipe, we use the same data as in the previous one, in order to highlight the differences in the approach and results.

Getting ready

For details on how to easily install R, please refer to the following resources:

- <https://cran.r-project.org/>
- <https://docs.anaconda.com/anaconda/user-guide/tasks/using-r-language/>

If you use conda as your package manager, the process of setting everything up can be greatly simplified. If you just install rpy2 using the conda install rpy2 command, the package manager will automatically install the latest version of R and some other required dependencies.

Before executing the following code, please make sure to run the code from the previous recipe to have the data available.

How to do it...

Execute the following steps to estimate a DCC-GARCH model in Python (using R):

1. Set up the connection between Python and R using rpy2:

```
%load_ext rpy2.ipython
```

2. Install the rmgarch R package and load it:

```
%%R  
  
install.packages('rmgarch', repos = "http://cran.us.r-project.org")  
library(rmgarch)
```

We only need to install the rmgarch package once. After doing so, you can safely comment out the line starting with `install.packages`.

3. Import the dataset into R:

```
%%R -i returns
print(head(returns))
```

Using the preceding command, we print the first five rows of the R `data.frame`:

	AAPL	GOOG	MSFT
2015-01-02 00:00:00	-0.951253138	-0.3020489	0.6673615
2015-01-05 00:00:00	-2.817148406	-2.0845731	-0.9195739
2015-01-06 00:00:00	0.009416247	-2.3177049	-1.4677364
2015-01-07 00:00:00	1.402220689	-0.1713264	1.2705295
2015-01-08 00:00:00	3.842214047	0.3153082	2.9418228

4. Define the model specification:

```
%%R

# define GARCH(1,1) model
univariate_spec <- ugarchspec(
    mean.model = list(armaOrder = c(0,0)),
    variance.model = list(garchOrder = c(1,1),
                           model = "sGARCH"),
    distribution.model = "norm"
)

# define DCC(1,1) model
n <- dim(returns)[2]
dcc_spec <- dccspec(
    uspec = multispec(replicate(n, univariate_spec)),
    dccOrder = c(1,1),
    distribution = "mvnorm"
)
```

5. Estimate the model:

```
%%R
dcc_fit <- dccfit(dcc_spec, data=returns)
dcc_fit
```

The following table contains the model's specification summary, estimated coefficients, as well as a selection of goodness-of-fit criteria:

* DCC GARCH Fit *	

Distribution	: mvnorm
Model	: DCC(1,1)
No. Parameters	: 17
[VAR GARCH DCC UncQ]	: [0+12+2+3]
No. Series	: 3
No. Obs.	: 1762
Log-Likelihood	: -8818.787
Av.Log-Likelihood	: -5
 Optimal Parameters	

	Estimate Std. Error t value Pr(> t)
[AAPL].mu	0.189285 0.037040 5.1102 0.000000
[AAPL].omega	0.176370 0.051204 3.4445 0.000572
[AAPL].alpha1	0.134726 0.026084 5.1651 0.000000
[AAPL].beta1	0.811601 0.029763 27.2691 0.000000
[GOOG].mu	0.125177 0.040152 3.1176 0.001823
[GOOG].omega	0.305000 0.163809 1.8619 0.062614
[GOOG].alpha1	0.183387 0.089046 2.0595 0.039449
[GOOG].beta1	0.715766 0.112531 6.3606 0.000000
[MSFT].mu	0.149371 0.030686 4.8677 0.000001
[MSFT].omega	0.269463 0.086732 3.1068 0.001891
[MSFT].alpha1	0.214566 0.052722 4.0698 0.000047
[MSFT].beta1	0.698830 0.055597 12.5695 0.000000
[Joint]dcca1	0.060145 0.016934 3.5518 0.000383
[Joint]dccb1	0.793072 0.059999 13.2180 0.000000
 Information Criteria	

Akaike	10.029
Bayes	10.082
Shibata	10.029
Hannan-Quinn	10.049

6. Calculate the five-step ahead forecasts:

```
forecasts <- dccforecast(dcc_fit, n.ahead = 5)
```

7. Access the forecasts:

```
%%R  
  
# conditional covariance matrix  
forecasts@mforecast$H  
# conditional correlation matrix  
forecasts@mforecast$R  
# proxy correlation process  
forecasts@mforecast$Q  
# conditional mean forecasts  
forecasts@mforecast$mu
```

The following image shows the five-step ahead forecasts of the conditional covariance matrix:

```
[[1]]  
, , 1  
  
[,1]      [,2]      [,3]  
[1,] 2.397337 1.086898 1.337702  
[2,] 1.086898 1.515434 1.145010  
[3,] 1.337702 1.145010 1.874023  
  
, , 2  
  
[,1]      [,2]      [,3]  
[1,] 2.445035 1.138809 1.367728  
[2,] 1.138809 1.667607 1.231062  
[3,] 1.367728 1.231062 1.981190  
  
, , 3  
  
[,1]      [,2]      [,3]  
[1,] 2.490173 1.184169 1.395189  
[2,] 1.184169 1.804434 1.308254  
[3,] 1.395189 1.308254 2.079076
```

```
, , 4

[,1]      [,2]      [,3]
[1,] 2.532888 1.224255 1.420526
[2,] 1.224255 1.927462 1.377669
[3,] 1.420526 1.377669 2.168484

, , 5

[,1]      [,2]      [,3]
[1,] 2.573311 1.259997 1.444060
[2,] 1.259997 2.038083 1.440206
[3,] 1.444060 1.440206 2.250150
```

We can now compare this forecast (the first step) to the one obtained using a simpler CCC-GARCH model. The values of the one-step ahead conditional covariance forecasts are very similar for CCC- and DCC-GARCH models.

How it works...

In this recipe, we used the same data as in the previous recipe, in order to compare the results of the CCC- and DCC-GARCH models. For more information on downloading the data, please refer to *Steps 1 to 4* in the previous recipe.

To work with Python and R at the same time, we used the `rpy2` library. In this recipe, we presented how to use the library in combination with Jupyter Notebook. For more details on how to use the library in a `.py` script, please refer to the official documentation. Also, we do not delve into the details of R code in general, as this is outside the scope of this book.

In *Step 1*, aside from loading any libraries, we also had to use the following magic command: `%load_ext rpy2.ipython`. It enabled us to run R code by adding `%%R` to the beginning of a cell in the Notebook. For that reason, please assume that any code block in this chapter is a separate Notebook cell (see the Jupyter Notebook in the accompanying GitHub repository for more information).

In *Step 2*, we had to install the required R dependencies. To do so, we used the `install.packages` function, and we specified the repository we wanted to use.

In *Step 3*, we moved the `pandas` DataFrame into the R environment. To do so, we passed the extra code `-i returns`, together with the `%%R` magic command. We could have imported the data in any of the ensuing steps.



When you want to move a Python object to R, do some manipulation/modeling, and move the final results back to Python, you can use the following syntax:
`%%R -i input_object -o output_object`.

In *Step 4*, we defined the DCC-GARCH model's specification. First, we defined the univariate GARCH specification (for conditional volatility estimation) using `ugarchspec`. This function comes from a package called `rugarch`, which is the framework for univariate GARCH modeling. By not specifying the ARMA parameters, we chose a constant mean model. For the volatility, we used a GARCH(1,1) model with normally distributed innovations. Secondly, we also specified the DCC model. To do so, we:

- Replicated the univariate specification for each returns series – in this case, three
- Specified the order of the DCC model—in this case, DCC(1,1)
- Specified the multivariate distribution—in this case, Multivariate Normal

We could see the summary of the specification by calling the `dcc_spec` object.

In *Step 5*, we estimated the model by calling the `dccfit` function with the specification and data as arguments. Afterward, we obtained five-step ahead forecasts by using the `dccforecast` function, which returned nested objects such as:

- `H`: the conditional covariance matrix
- `R`: the conditional correlation matrix
- `Q`: the proxy process for the correlation matrix
- `mu`: the conditional mean

Each one of them contained five-step ahead forecasts, stored in lists.

There's more...

In this section, we would also like to go over a few more details on estimating GARCH models.

Estimation details

In the first step of estimating the DCC-GARCH model, we can additionally use an approach called **variance targeting**. The idea is to reduce the number of parameters we need to estimate in the GARCH model.

To do so, we can slightly modify the GARCH equation. The original equation runs as follows:

$$\sigma_t^2 = \omega + \sum_{i=1}^q \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^p \beta_i \sigma_{t-i}^2$$

Unconditional volatility is defined as:

$$\bar{\sigma} = \omega / (1 - \alpha - \beta)$$

We can now plug it into the GARCH equation and produce the following:

$$\sigma_t^2 = \bar{\sigma} (1 - \alpha - \beta) + \sum_{i=1}^q \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^p \beta_i \sigma_{t-i}^2$$

In the last step, we replace the unconditional volatility with the sample variance of the returns:

$$\hat{\sigma} = \frac{1}{T} \sum_{t=1}^T \epsilon_t^2$$

By doing so, we have one less parameter to estimate for each GARCH equation. Also, the unconditional variance implied by the model is guaranteed to be equal to the unconditional sample variance. To use variance targeting in practice, we add an extra argument to the `ugarchspec` function call: `ugarchspec(..., variance.targeting = TRUE)`.

Univariate and multivariate GARCH models

It is also worth mentioning that `rugarch` and `rmgarch` work nicely together, as they were both developed by the same author and created as a single go-to framework for estimating GARCH models in R. We have already gained some experience with this when we used the `ugarchspec` function in the first step of estimating the DCC-GARCH model. There is much more to discover in terms of that package.

Parallelizing the estimation of multivariate GARCH models

Lastly, the estimation process of the DCC-GARCH model can be easily parallelized, with the help of the `parallel` R package.

To potentially speed up computations with parallelization, we reused the majority of the code from this recipe and added a few extra lines. First, we had to set up a cluster by using `makePSOCKcluster` from the `parallel` package and indicated that we would like to use three cores. Then, we defined the parallelizable specification using `multifit`. Lastly, we fitted the DCC-GARCH model. The difference here, compared to the previously used code, is that we additionally passed the `fit` and `cluster` arguments to the function call. When we are done with the estimation, we stop the cluster. You can find the entire snippet below:

```
%%R

# parallelized DCC-GARCH(1,1)
library("parallel")

# set up the cluster
cl <- makePSOCKcluster(3)

# define parallelizable specification
parallel_fit <- multifit(multispec(replicate(n, univariate_spec)),
                           returns,
                           cluster = cl)
```

```
# fit the DCC-GARCH model
dcc_fit <- dccfit(dcc_spec,
                    data = returns,
                    fit.control = list(eval.se = TRUE),
                    fit = parallel_fit,
                    cluster = cl)

# stop the cluster
stopCluster(cl)
```

Using the preceding code, we can significantly speed up the estimation of the DCC-GARCH model. The improvement in performance is mostly visible when dealing with large volumes of data. Also, the approach of using the `parallel` package together with `multifit` can be used to speed up the calculations of various GARCH and ARIMA models from the `rugarch` and `rmgarch` packages.

See also

Additional resources:

- Engle, R.F., 2002. “Dynamic Conditional Correlation: A Simple Class of Multivariate Generalized Autoregressive Conditional Heteroskedasticity Models,” *Journal of Business and Economic Statistics*, 20(3): 339–350: <https://doi.org/10.1198/073500102288618487>
- Ghalanos, A. (2019). The `rmgarch` models: Background and properties. (Version 1.3-0): https://cran.r-project.org/web/packages/rmgarch/vignettes/The_rmgarch_models.pdf
- `rpy2`'s documentation: <https://rpy2.github.io/>

Summary

Volatility modeling and forecasting have attracted significant attention in recent years, largely due to their importance in financial markets. In this chapter, we have covered the practical application of GARCH models (both univariate and multivariate) to volatility forecasting. By knowing how to model volatility using GARCH class models, we can use more accurate volatility forecasts to replace the naïve estimates in many practical use cases, for example, risk management, volatility trading, and derivatives valuation.

We have focused on GARCH models due to their ability to capture volatility clustering. However, there are other approaches to volatility modeling. For example, regime-switching models assume that there are certain repeating patterns (regimes) in data. Therefore, we should be able to predict future states by using parameter estimates based on past observations.

10

Monte Carlo Simulations in Finance

Monte Carlo simulations are a class of computational algorithms that use repeated random sampling to solve any problems that have a probabilistic interpretation. In finance, one of the reasons they gained popularity is that they can be used to accurately estimate integrals. The main idea of Monte Carlo simulations is to produce a multitude of sample paths (possible scenarios/outcomes), often over a given period of time. The horizon is then split into a specified number of time steps and the process of doing so is called discretization. Its goal is to approximate the continuous time in which the pricing of financial instruments happens.

The results from all of these simulated sample paths can be used to calculate metrics such as the percentage of times an event occurred, the average value of an instrument at the last step, and so on. Historically, the main problem with the Monte Carlo approach was that it required heavy computational power to calculate all of the considered scenarios. Nowadays, this is becoming less of a problem as we can run fairly advanced simulations on a desktop computer or a laptop, and if we run out of computing power, we can use cloud computing and its more powerful processors.

By the end of this chapter, we will have seen how we can use Monte Carlo methods in various scenarios and tasks. In some of them, we will create the simulations from scratch, while in others, we will use modern Python libraries to make the process even easier. Due to the method's flexibility, Monte Carlo is one of the most important techniques in computational finance. It can be adapted to various problems, such as pricing derivatives with no closed-form solution (American/exotic options), valuation of bonds (for example, a zero-coupon bond), estimating the uncertainty of a portfolio (for example, by calculating Value-at-Risk and Expected Shortfall), and carrying out stress tests in risk management. We will show you how to solve some of these problems in this chapter.

In this chapter, we cover the following recipes:

- Simulating stock price dynamics using a geometric Brownian motion
- Pricing European options using simulations
- Pricing American options with Least Squares Monte Carlo
- Pricing American options using QuantLib
- Pricing barrier options
- Estimating Value-at-Risk using Monte Carlo

Simulating stock price dynamics using a geometric Brownian motion

Simulating stock prices plays a crucial role in the valuation of many derivatives, most notably options. Due to the randomness in the price movement, these simulations rely on **stochastic differential equations (SDEs)**. A stochastic process is said to follow a **geometric Brownian motion (GBM)** when it satisfies the following SDE:

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

Here, we have the following:

- S_t —Stock price
- μ —The drift coefficient, that is, the average return over a given period or the instantaneous expected return
- σ —The diffusion coefficient, that is, how much volatility is in the drift
- W_t —The Brownian motion
- d —This symbolizes the change in the variable over the considered time increment, while dt is the change in time

We will not investigate the properties of the **Brownian motion** in too much depth, as it is outside the scope of this book. Suffice to say, Brownian increments are calculated as a product of a Standard Normal random variable ($rv \sim N(0,1)$) and the square root of the time increment.

Another way to say this is that the Brownian increment comes from $rv \sim N(0, t)$, where t is the time increment. We obtain the Brownian path by taking the cumulative sum of the Brownian increments.

The SDE mentioned above is one of the few that has a closed-form solution:

$$S(t) = S_0 e^{(\mu - \frac{1}{2}\sigma^2)t + \sigma W_t}$$

Where $S_0 = S(0)$ is the initial value of the process, which in this case is the initial price of a stock. The preceding equation presents the relationship between the stock price at time t and the initial stock price.

For simulations, we can use the following recursive formula:

$$S(t_{i+1}) = S(t_i) \exp \left((\mu - \frac{1}{2} \sigma^2)(t_{i+1} - t_i) + \sigma \sqrt{t_{i+1} - t_i} Z_{i+1} \right)$$

Where Z_i is a Standard Normal random variable and $i = 0, 1, \dots, T-1$ is the time index. This specification is possible because the increments of W are independent and normally distributed. Please refer to Euler's discretization for a better understanding of the formula's origin.



A GBM is a process that does not account for mean-reversion and time-dependent volatility. That is why it is often used for stocks and not for bond prices, which tend to display long-term reversion to the face value.

In this recipe, we use Monte Carlo methods and a GBM to simulate IBM's stock prices one month ahead—using data from 2021, we will simulate the possible paths over January 2022.

How to do it...

Execute the following steps to simulate IBM's stock prices one month ahead:

1. Import the libraries:

```
import numpy as np
import pandas as pd
import yfinance as yf
```

2. Download IBM's stock prices from Yahoo Finance:

```
df = yf.download("IBM",
                 start="2021-01-01",
                 end="2022-01-31",
                 adjusted=True)
```

3. Calculate and plot the daily returns:

```
returns = df["Adj Close"].pct_change().dropna()
returns.plot(title="IBM's returns")
```

Running the snippet produces the following plot:

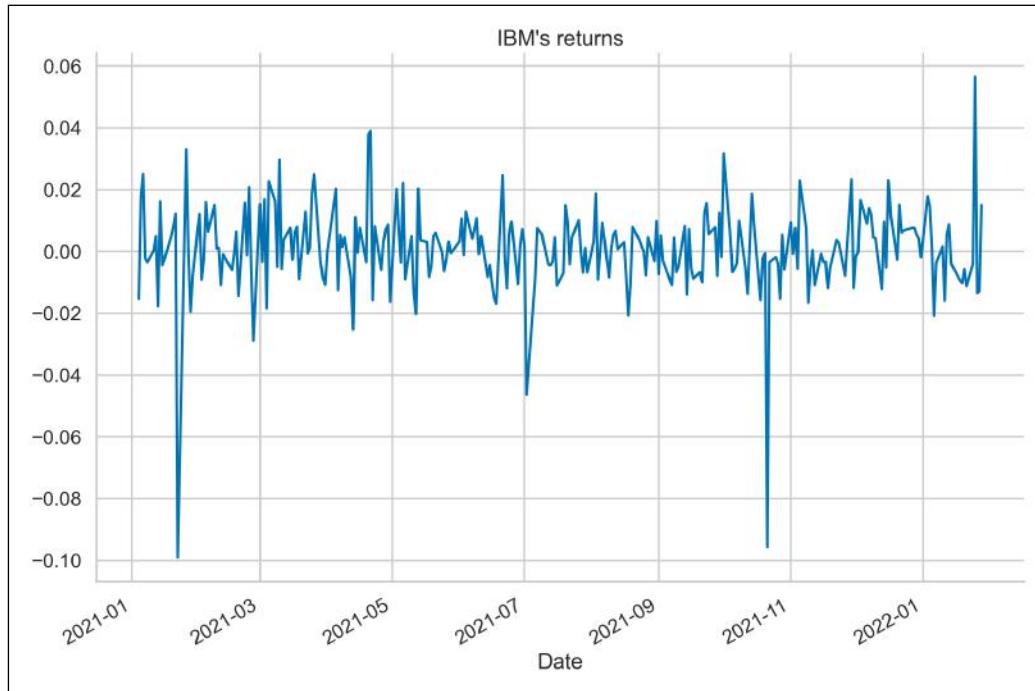


Figure 10.1: IBM's simple returns

4. Split the data into training and test sets:

```
train = returns["2021"]
test = returns["2022"]
```

5. Specify the parameters of the simulation:

```
T = len(test)
N = len(test)
S_0 = df.loc[train.index[-1], "Adj Close"]
N_SIM = 100
mu = train.mean()
sigma = train.std()
```

6. Define the function used for the simulations:

```
def simulate_gbm(s_0, mu, sigma, n_sims, T, N,
                  random_seed=42):
    np.random.seed(random_seed)

    dt = T/N
    dW = np.random.normal(scale=np.sqrt(dt), size=(n_sims, N))
    W = np.cumsum(dW, axis=1)

    time_step = np.linspace(dt, T, N)
    time_steps = np.broadcast_to(time_step, (n_sims, N))

    S_t = (
        s_0 * np.exp((mu - 0.5 * sigma**2) * time_steps + sigma * W)
    )
    S_t = np.insert(S_t, 0, s_0, axis=1)

    return S_t
```

7. Run the simulations and store the results in a DataFrame:

```
gbm_simulations = simulate_gbm(S_0, mu, sigma, N_SIM, T, N)
sim_df = pd.DataFrame(np.transpose(gbm_simulations),
                      index=train.index[-1:].union(test.index))
```

8. Create a DataFrame with the average value for each time step and the corresponding actual stock price:

```
res_df = sim_df.mean(axis=1).to_frame()
res_df = res_df.join(df["Adj Close"])
res_df.columns = ["simulation_average", "adj_close_price"]
```

9. Plot the results of the simulation:

```
ax = sim_df.plot(
    alpha=0.3, legend=False, title="Simulation's results"
)
res_df.plot(ax=ax, color = ["red", "blue"])
```

In Figure 10.2, we observe that the predicted stock prices (the averages of the simulations for each time step) exhibit a slightly positive trend. That could be attributed to the positive drift term $\mu = 0.07\%$. However, we should take that conclusion with a pinch of salt given the very small number of simulations.

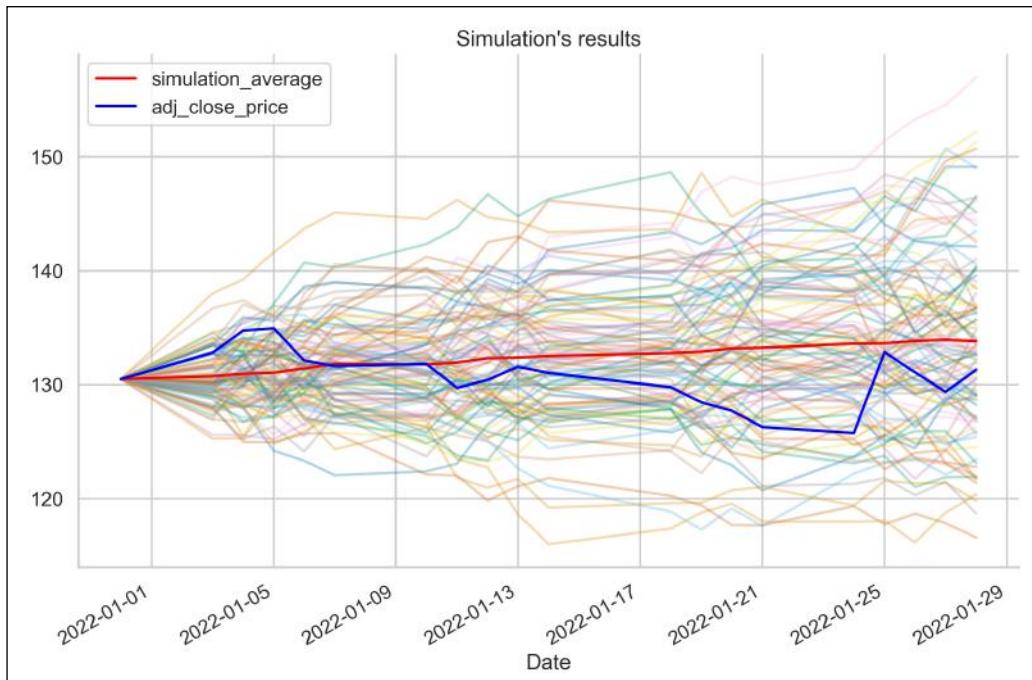


Figure 10.2: The simulated paths together with their average

Bear in mind that such a visualization is only feasible for a reasonable number of sample paths. In real-life cases, we want to use significantly more sample paths than 100. The general approach to Monte Carlo simulations is that having more sample paths leads to more accurate/reliable results.

How it works...

In Steps 2 and 3, we downloaded IBM's stock prices and calculated simple returns. In the next step, we divided the data into the training and test sets. While there is no explicit training of any model here, we used the training set to calculate the average and standard deviation of the returns. We then used those values as the drift (μ) and diffusion (σ) coefficients for our simulations. Additionally, in Step 5, we defined the following parameters:

- T : Forecasting horizon; in this case, the number of days in the test set.
- N : Number of time increments in the forecasting horizon. For our simulation, we keep $N = T$.
- S_0 : Initial price. For this simulation, we use the last observation from the training set.

- `N_SIM`: Number of simulated paths.



Monte Carlo simulations use a process called discretization. The idea is to approximate the continuous pricing of financial assets by splitting the considered time horizon into a large number of discrete intervals. That is why, except for considering the forecasting horizon, we also need to indicate the number of time increments to fit into the horizon.

In *Step 6*, we defined the function for running the simulations. It is good practice to define a function/class for such a problem, as it will also come in handy in the following recipes. The function executes the following steps:

1. Defines the time increment (`dt`) and the Brownian increments (`dW`). In the matrix of Brownian increments (size: `N_SIM × N`), each row describes one sample path.
2. Calculates the Brownian paths (`W`) by running a cumulative sum (`np.cumsum`) over the rows.
3. Creates a matrix containing the time steps (`time_steps`). To do so, we created an array of evenly spaced values within an interval (the horizon of the simulation). For that, we used the `np.linspace` function. Afterward, we broadcasted the array to the intended shape using `np.broadcast_to`.
4. Calculates the stock price at each point in time using the closed-form formula.
5. Inserts the initial value into the first position of each row.



There was no explicit need to broadcast the vector containing time steps. It would have been done automatically to match the required dimensions (the dimension of `W`). By doing it manually, we get more control over what we are doing, which makes the code easier to debug. We should also be aware that in languages such as R, there is no automatic broadcasting.

In the function's definition, we can recognize the drift as `(mu - 0.5 * sigma ** 2) * time_steps` and the diffusion as `sigma * W`. Additionally, while defining this function, we followed the vectorized approach. By doing so, we avoided writing any `for` loops, which would be inefficient in the case of large simulations.



For reproducible results, use `np.random.seed` before simulating the paths.

In *Step 7*, we ran the simulations and stored the outcome (sample paths) in a `DataFrame`. While doing so, we transposed the data so that we had one path per column, which simplifies using the `plot` method of the `pandas DataFrame`. To have the appropriate index, we used the `union` method of a `DatetimeIndex` to join the index of the last observation from the training set and the indices from the test set.

In Step 8, we calculated the predicted stock price as the average value of all the simulations for each point of time and stored those results in a DataFrame. Then, we also joined the actual stock prices for each date.

In the last step, we visualized the simulated sample paths. While visualizing the simulated paths, we chose `alpha=0.3` to make the lines transparent. By doing so, it is easier to see the two lines representing the predicted (average) path and the actual one.

There's more...

There are some statistical methods that make working with Monte Carlo simulations easier (higher accuracy, faster computations). One of them is a variance reduction method called **antithetic variates**. In this approach, we try to reduce the variance of the estimator by introducing negative dependence between pairs of random draws. This translates into the following: when creating sample paths, for each $[\epsilon_1, \dots, \epsilon_t]$, we also take the antithetic values, that is, $[-\epsilon_1, \dots, -\epsilon_t]$.

The advantages of this approach are:

- Reduction (by half) of the number of Standard Normal samples to be drawn in order to generate N paths
- Reduction of the sample path variance, while at the same time improving the accuracy

We implemented this approach in the improved `simulate_gbm` function. Additionally, we made the function shorter by putting the majority of the calculations into one line.

Before we implemented these changes, we timed the initial version of the function:

```
%timeit gbm_simulations = simulate_gbm(S_0, mu, sigma, N_SIM, T, N)
```

The score was:

```
71 µs ± 126 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The new function is defined as follows:

```
def simulate_gbm(s_0, mu, sigma, n_sims, T, N, random_seed=42,
                  antithetic_var=False):
    np.random.seed(random_seed)

    # time increment
    dt = T/N

    # Brownian
    if antithetic_var:
        dW_ant = np.random.normal(scale = np.sqrt(dt),
                                   size=(int(n_sims/2), N + 1))
        dW = np.concatenate((dW_ant, -dW_ant), axis=0)
```

```

else:
    dW = np.random.normal(scale = np.sqrt(dt),
                          size=(n_sims, N + 1))

    # simulate the evolution of the process
    S_t = s_0 * np.exp(np.cumsum((mu - 0.5*sigma**2)*dt + sigma*dW,
                                  axis=1))
    S_t[:, 0] = s_0

return S_t

```

First, we run the simulations without antithetic variables:

```
%timeit gbm_simulations = simulate_gbm(S_0, mu, sigma, N_SIM, T, N)
```

Which scores:

```
50.3 µs ± 275 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Then, we run the simulations with antithetic variables:

```
%timeit gbm_simulations = simulate_gbm(S_0, mu, sigma, N_SIM, T, N, antithetic_
var=True)
```

Which scores:

```
38.2 µs ± 623 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

We succeeded in making the function faster. If you are interested in pure performance, these simulations can be further expedited using Numba, Cython, or multiprocessing.



Other possible variance reduction techniques include control variates and common random numbers.

See also

In this recipe, we have shown how to simulate stock prices using a geometric Brownian motion. However, there are other stochastic processes that could be used as well, some of which are:

- Jump-diffusion model: Merton, R. “Option Pricing When the Underlying Stock Returns Are Discontinuous,” *Journal of Financial Economics*, 3, 3 (1976): 125–144
- Square-root diffusion model: Cox, John, Jonathan Ingersoll, and Stephen Ross , “A theory of the term structure of interest rates,” *Econometrica*, 53, 2 (1985): 385–407
- Stochastic volatility model: Heston, S. L., “A closed-form solution for options with stochastic volatility with applications to bond and currency options,” *The Review of Financial Studies*, 6(2): 327-343.

Pricing European options using simulations

Options are a type of derivative instrument because their price is linked to the price of the underlying security, such as stock. Buying an options contract grants the right, but not the obligation, to buy or sell an underlying asset at a set price (known as a strike) on/before a certain date. The main reason for the popularity of options is because they hedge away exposure to an asset's price moving in an undesirable way.

In this recipe we will focus on one type of option, that is, European options. A **European call/put option** gives us the right (but again, no obligation) to buy/sell a certain asset on a certain expiry date (commonly denoted as T).

There are many possible ways of option valuation, for example, using:

- Analytical formulas (only some kinds of options have those)
- Binomial tree approach
- Finite differences
- Monte Carlo simulations

European options are an exception in the sense that there exists an analytical formula for their valuation, which is not the case for more advanced derivatives, such as American or exotic options.

To price options using Monte Carlo simulations, we use risk-neutral valuation, under which the fair value of a derivative is the expected value of its future payoff(s). In other words, we assume that the option premium grows at the same rate as the risk-free rate, which we use for discounting to the present value. For each of the simulated paths, we calculate the option's payoff at maturity, take the average of all the paths, and discount it to the present value.

In this recipe, we show how to code the closed-form solution of the Black-Scholes model and then use the Monte Carlo simulation approach. For simplicity, we use fictitious input data, but real-life data could be used analogically.

How to do it...

Execute the following steps to price European options using the analytical formula and Monte Carlo simulations:

1. Import the libraries:

```
import numpy as np
from scipy.stats import norm
from chapter_10_utils import simulate_gbm
```

In this recipe, we use the `simulate_gbm` function we have defined in the previous recipe. For our convenience, we store it in a separate .py script, from which we can import it.

2. Define the option's parameters for the valuation:

```
S_0 = 100
K = 100
r = 0.05
sigma = 0.50
T = 1
N = 252
dt = T / N
N_SIMS = 1_000_000
discount_factor = np.exp(-r * T)
```

3. Prepare the valuation function using the analytical solution:

```
def black_scholes_analytical(S_0, K, T, r, sigma, type="call"):
    d1 = (
        np.log(S_0 / K) + (r + 0.5*sigma**2) * T) / (sigma*np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    if type == "call":
        N_d1 = norm.cdf(d1, 0, 1)
        N_d2 = norm.cdf(d2, 0, 1)
        val = S_0 * N_d1 - K * np.exp(-r * T) * N_d2
    elif type == "put":
        N_d1 = norm.cdf(-d1, 0, 1)
        N_d2 = norm.cdf(-d2, 0, 1)
        val = K * np.exp(-r * T) * N_d2 - S_0 * N_d1
    else:
        raise ValueError("Wrong input for type!")
    return val
```

4. Evaluate a call option using the specified parameters:

```
black_scholes_analytical(S_0=S_0, K=K, T=T,
                           r=r, sigma=sigma,
                           type="call")
```

The price of a European call option with the specified parameters is 21.7926.

5. Simulate the stock path using the `simulate_gbm` function:

```
gbm_sims = simulate_gbm(s_0=S_0, mu=r, sigma=sigma,
                           n_sims=N_SIMS, T=T, N=N)
```

6. Calculate the option's premium:

```
premium = (
    discount_factor * np.mean(np.maximum(0, gbm_sims[:, -1] - K))
)
premium
```

The calculated option premium is 21.7562. Please bear in mind that we are using a fixed random seed in the `simulate_gbm` function to obtain reproducible results. In general, whenever we are dealing with simulations, we can expect some degree of randomness in the results.

Here, we can see that the option premium that we calculated using Monte Carlo simulations is close to the one from a closed-form solution of the Black-Scholes model. To increase the accuracy of the simulation, we could increase the number of simulated paths (using the `N_SIMS` parameter).

How it works...

In *Step 2*, we defined the parameters that we used for this recipe:

- `S_0`: Initial stock price
- `K`: Strike price, that is, the one we can buy/sell for at maturity
- `r`: Annual risk-free rate
- `sigma`: Underlying stock volatility (annualized)
- `T`: Time until maturity in years
- `N`: Number of time increments for simulations
- `N_SIMS`: Number of simulated sample paths
- `discount_factor`: Discount factor, which is used to calculate the present value of the future payoff

In *Step 3*, we defined a function for calculating the option premium using the closed-form solution to the Black-Scholes model (for non-dividend-paying stocks). We used it in *Step 4* to calculate the benchmark for the Monte Carlo simulations.

The analytical solutions to the call and put options are defined as follows:

$$C(S_t, t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)}$$

$$P(S_t, t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}} \left[\ln\left(\frac{S_t}{K}\right) + (r + \frac{\sigma^2}{2})(T-t) \right]$$

$$d_2 = d_1 - \sigma\sqrt{T-t}$$

Where `N()` stands for the **cumulative distribution function (CDF)** of the Standard Normal distribution and $T - t$ is the time to maturity expressed in years. Equation 1 represents the formula for the price of a European call option, while equation 2 represents the price of the European put option. Informally, the two terms in equation 1 can be thought of as:

- The current price of the stock, weighted by the probability of exercising the option to buy the stock ($N(d_1)$)—in other words, what we could receive
- The discounted price of exercising the option (strike), weighted by the probability of exercising the option ($N(d_2)$)—in other words, what we are going to pay

In *Step 5*, we used the GBM simulation function from the previous recipe to obtain 1,000,000 possible paths of the underlying asset. To calculate the option premium, we only looked at the terminal values, and for each path, calculated the payoff as follows:

- $\max(S_T - K, 0)$ for the call option
- $\max(K - S_T, 0)$ for the put option

In *Step 6*, we took the average of the payoffs and discounted it to present the value by using the discount factor.

There's more...

Improving the valuation function using Monte Carlo simulations

In the previous steps, we showed how to reuse the GBM simulation to calculate the European call option premium. However, we can make the calculations faster, as in the case of European options we are only interested in the terminal stock price. The intermediate steps do not matter. That is why we only need to simulate the price at time T and use these values to calculate the expected payoff. We show how to do this by using an example of a European put option with the same parameters as we used before.

We start by calculating the option premium using the analytical formula:

```
black_scholes_analytical(S_0=S_0, K=K, T=T, r=r, sigma=sigma, type="put")
```

The calculated option premium is 16.9155.

Then, we define the modified simulation function, which only looks at the terminal values of the simulation paths:

```
def european_option_simulation(S_0, K, T, r, sigma, n_sims,
                               type="call", random_seed=42):
    np.random.seed(random_seed)
    rv = np.random.normal(0, 1, size=n_sims)
    S_T = S_0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * rv)

    if type == "call":
        payoff = np.maximum(0, S_T - K)
    elif type == "put":
        payoff = np.maximum(0, K - S_T)
    else:
        raise ValueError("Wrong input for type!")

    premium = np.mean(payoff) * np.exp(-r * T)
    return premium
```

Then, we run the simulations:

```
european_option_simulation(S_0, K, T, r, sigma, N_SIMS, type="put")
```

The resulting value is 16.9482, which is close to the previous value. Further increasing the number of simulated paths should increase the accuracy of the valuation.

Measuring price sensitivity with the Greeks

While talking about the valuation of options, it is also worthwhile to mention the famous Greeks—quantities representing the sensitivity of the price of financial derivatives to a change in one of the underlying parameters. The name comes from the fact that those sensitivities are most commonly denoted using the letters of the Greek alphabet. The following are the five most popular sensitivities:

- Delta (Δ): The sensitivity of the theoretical option value with respect to the changes in the underlying asset's price
- Vega (∇): The sensitivity of the theoretical option value with respect to the volatility of the underlying asset
- Theta (Θ): The sensitivity of the theoretical option value with respect to the option's time to maturity
- Rho (ρ): The sensitivity of the theoretical option value with respect to the interest rates
- Gamma (Γ): This is an example of a second-order Greek as it represents the sensitivity of the option's delta (Δ) with respect to the changes in the underlying asset's price

The following table shows how the Greeks of European call and put options are expressed in terms of the values we have already used for calculating the option's premium using the analytical formulas:

	What	Calls	Puts
delta	$\frac{\partial C}{\partial S}$	$N(d_1)$	$-N(-d_1) = N(d_1) - 1$
gamma	$\frac{\partial^2 C}{\partial S^2}$		$\frac{N'(d_1)}{S\sigma\sqrt{T-t}}$
vega	$\frac{\partial C}{\partial \sigma}$		$SN'(d_1)\sqrt{T-t}$
theta	$\frac{\partial C}{\partial t}$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}N(d_2)$	$-\frac{SN'(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}N(-d_2)$
rho	$\frac{\partial C}{\partial r}$	$K(T-t)e^{-r(T-t)}N(d_2)$	$-K(T-t)e^{-r(T-t)}N(-d_2)$

The $N'()$ symbol represents the **probability density function (PDF)** of the Standard Normal distribution. As you can see, the Greeks are actually partial derivatives of some model price (in this case, European call or put options) with respect to one of the model's parameters. We should also keep in mind that the Greeks differ by model.

Pricing American options with Least Squares Monte Carlo

In this recipe, we learn how to value American options. The key difference between European and American options is that the latter can be exercised at any time before and including the maturity date—basically, whenever the underlying asset's price moves favorably for the option holder.

This behavior introduces additional complexity to the valuation and there is no closed-form solution to this problem. When using Monte Carlo simulations, we cannot only look at the terminal value on each sample path, as the option's exercise can happen anywhere along the path. That is why we need to employ a more sophisticated approach called **Least Squares Monte Carlo (LSMC)**, which was introduced by Longstaff and Schwartz (2001).

First of all, the time axis spanning $[0, T]$ is discretized into a finite number of equally spaced intervals and the early exercise can happen only at those particular time steps. Effectively, the American option is approximated by a Bermudan one. For any time step t , the early exercise is performed in case the payoff from the immediate exercise is larger than the continuation value.

This is expressed by the following formula:

$$V_t(s) = \max(h_t(s), C_t(s))$$

Here, $h_t(s)$ stands for the option's payoff (also called the option's inner value, calculated as in the case of European options) and $C_t(s)$ is the continuation value of the option, which is defined as:

$$C_t(s) = E_t^Q[e^{-rdt}V_{t+dt}(S_{t+dt})|S_t = s]$$

Here, r is the risk-free rate, dt is the time increment, and $E_t^Q(\dots | S_t = s)$ is the risk-neutral expectation given the underlying price. The continuation value is basically the expected payoff from not exercising the option at a given time.

When using Monte Carlo simulations, we can define the continuation value $e^{-rdt}V_{t+dt,i}$ for each path i and time t . Using this value directly is not possible as this would imply perfect foresight. That is why the LSMC algorithm uses linear regression to estimate the expected continuation value. In the algorithm, we regress the discounted future values (obtained from keeping the option) onto a set of basis functions of the spot price (time t price). The simplest way to approach this is to use an x -degree polynomial regression. Other options for the basis functions include Legendre, Hermite, Chebyshev, Gegenbauer, or Jacobi polynomials.

We iterate this algorithm backward (from time $T-1$ to 0) and at the last step take the average discounted value as the option premium. The premium of a European option represents the lower bound to the American option's premium. The difference is usually called the early exercise premium.

How to do it...

Execute the following steps to price American options using the Least Squares Monte Carlo method:

1. Import the libraries:

```
import numpy as np
from chapter_10_utils import (simulate_gbm,
                               black_scholes_analytical,
                               lsmc_american_option)
```

2. Define the option's parameters:

```
S_0 = 36
K = 40
r = 0.06
sigma = 0.2
T = 1 # 1 year
N = 50
dt = T / N
N_SIMS = 10 ** 5
discount_factor = np.exp(-r * dt)
OPTION_TYPE = "put"
POLY_DEGREE = 5
```

3. Simulate the stock prices using a GBM:

```
gbm_sims = simulate_gbm(s_0=S_0, mu=r, sigma=sigma,
                         n_sims=N_SIMS, T=T, N=N)
```

4. Calculate the payoff matrix:

```
payoff_matrix = np.maximum(K - gbm_sims, np.zeros_like(gbm_sims))
```

5. Define the value matrix and fill in the last column (time T):

```
value_matrix = np.zeros_like(payoff_matrix)
value_matrix[:, -1] = payoff_matrix[:, -1]
```

6. Iteratively calculate the continuation value and the value vector in the given time:

```
for t in range(N - 1, 0, -1):
    regression = np.polyfit(
        gbm_sims[:, t],
        value_matrix[:, t + 1] * discount_factor,
        POLY_DEGREE
    )
```

```

continuation_value = np.polyval(regression, gbm_sims[:, t])
value_matrix[:, t] = np.where(
    payoff_matrix[:, t] > continuation_value,
    payoff_matrix[:, t],
    value_matrix[:, t + 1] * discount_factor
)

```

7. Calculate the option's premium:

```

option_premium = np.mean(value_matrix[:, 1] * discount_factor)
option_premium

```

The premium on the specified American put option is 4.465.

8. Calculate the premium of a European put with the same parameters:

```

black_scholes_analytical(S_0=S_0, K=K, T=T, r=r, sigma=sigma,
                           type="put")

```

The price of the European put option with the same parameters is 3.84.

9. As an extra check, calculate the prices of the American and European call options:

```

european_call_price = black_scholes_analytical(
    S_0=S_0, K=K, T=T, r=r, sigma=sigma
)
american_call_price = lsmc_amERICAN_option(
    S_0=S_0, K=K, T=T, N=N, r=r,
    sigma=sigma, n_sims=N_SIMS,
    option_type="call",
    poly_degree=POLY_DEGREE
)
print(f"European call's price: {european_call_price:.3f}")
print(f"American call's price: {american_call_price:.3f}")

```

The price of the European call is 2.17, while the American call's price (using 100,000 simulations) is 2.10.

How it works...

In *Step 2*, we once again defined the parameters of the considered American option. For comparison's sake, we took the same values that Longstaff and Schwartz (2001) did. In *Step 3*, we simulated the stock's evolution using the `simulate_gbm` function from the previous recipe. Afterward, we calculated the payoff matrix of the put option using the same formula that we used for the European options.

In *Step 5*, we prepared the matrix of option values over time, which we defined as a matrix of zeros of the same size as the payoff matrix. We filled the last column of the value matrix with the last column of the payoff matrix, as at the last step there are no further computations to carry out—the payoff is equal to the European option.

In Step 6, we ran the backward part of the algorithm from time $T-1$ to 0. At each of these steps, we estimated the expected continuation value as a cross-sectional linear regression. We fitted the 5th-degree polynomial to the data using `np.polyfit`.

Then, we evaluated the polynomial at specific values (using `np.polyval`), which is the same as getting the fitted values from a linear regression. We compared the expected continuation value to the payoff to see if the option should be exercised. If the payoff was higher than the expected value from continuation, we set the value to the payoff. Otherwise, we set it to the discounted one-step-ahead value. We used `np.where` for this selection.



It is also possible to use `scikit-learn` for the polynomial fit. To do so, you need to combine `LinearRegression` with `PolynomialFeatures`.

In Step 7 of the algorithm, we obtained the option premium by taking the average value of the discounted $t = 1$ value vector.

In the last two steps, we carried out some sanity checks. First, we calculated the premium of a European put with the same parameters. Second, we repeated all the steps to get the premiums of American and European call options with the same parameters. To make this easier, we put the entire algorithm for LSMC into one function, which is available in this book's GitHub repository.

For the call option, the premium on the American and European options should be equal, as it is never optimal to exercise the option when there are no dividends. Our results are very close, but we can obtain a more accurate price by increasing the number of simulated sample paths.

In principle, the Longstaff-Schwartz algorithm should underprice American options because the approximation of the continuation value by the basis functions is just that, an approximation. As a consequence, the algorithm will not always make the correct decision about exercising the option. This, in turn, means that the option's value will be lower than in the case of the optimal exercise.

See also

Additional resources are available here:

- Longstaff, F. A., & Schwartz, E. S. 2001. “Valuing American options by simulation: a simple least-squares approach,” *The Review of Financial Studies*, 14(1): 113-147
- Broadie, M., Glasserman, P., & Jain, G. 1997. “An alternative approach to the valuation of American options using the stochastic tree method. Enhanced Monte Carlo estimates for American option prices,” *Journal of Derivatives*, 5: 25-44.

Pricing American options using QuantLib

In the previous recipe, we showed how to manually code the Longstaff-Schwartz algorithm. However, we can also use already existing frameworks for the valuation of derivatives. One of the most popular ones is QuantLib. It is an open-source C++ library that provides tools for the valuation of financial instruments. By using **Simplified Wrapper and Interface Generator (SWIG)**, it is possible to use QuantLib from Python (and some other programming languages, such as R or Julia). In this recipe, we show how to price the same American put option that we priced in the previous recipe, but the library itself has many more interesting features to explore.

Getting ready

Execute *Step 2* from the previous recipe to have the parameters of the American put option that we will value using QuantLib.

How to do it...

Execute the following steps to price American options using QuantLib:

1. Import the library:

```
import QuantLib as ql
```

2. Specify the calendar and the day-counting convention:

```
calendar = ql.UnitedStates()
day_counter = ql.ActualActual()
```

3. Specify the valuation date and the expiry date of the option:

```
valuation_date = ql.Date(1, 1, 2020)
expiry_date = ql.Date(1, 1, 2021)
ql.Settings.instance().evaluationDate = valuation_date
```

4. Define the option type (call/put), type of exercise (American), and payoff:

```
if OPTION_TYPE == "call":
    option_type_ql = ql.Option.Call
elif OPTION_TYPE == "put":
    option_type_ql = ql.Option.Put

exercise = ql.AmericanExercise(valuation_date, expiry_date)
payoff = ql.PlainVanillaPayoff(option_type_ql, K)
```

5. Prepare the market-related data:

```
u = ql.SimpleQuote(S_0)
r = ql.SimpleQuote(r)
sigma = ql.SimpleQuote(sigma)
```

6. Specify the market-related curves:

```
underlying = ql.QuoteHandle(u)
volatility = ql.BlackConstantVol(0, ql.TARGET(),
                                 ql.QuoteHandle(sigma),
                                 day_counter)
risk_free_rate = ql.FlatForward(0, ql.TARGET(),
                                ql.QuoteHandle(r),
                                day_counter)
```

7. Plug the market-related data into the Black-Scholes process:

```
bs_process = ql.BlackScholesProcess(
    underlying,
    ql.YieldTermStructureHandle(risk_free_rate),
    ql.BlackVolTermStructureHandle(volatility),
)
```

8. Instantiate the Monte Carlo engine for the American options:

```
engine = ql.MCAmericanEngine(
    bs_process, "PseudoRandom", timeSteps=N,
    polynomOrder=POLY_DEGREE,
    seedCalibration=42,
    requiredSamples=N_SIMS
)
```

9. Instantiate the option object and set its pricing engine:

```
option = ql.VanillaOption(payoff, exercise)
option.setPricingEngine(engine)
```

10. Calculate the option's premium:

```
option_premium_ql = option.NPV()
option_premium_ql
```

The value of the American put option is 4.457.

How it works...

Since we wanted to compare the results we obtained with those in the previous recipes, we used the same problem setup as we did there. For brevity, we will not look at all the code here, but we should run *Step 2* from the previous recipe.

In *Step 2*, we specified the calendar and the day-counting convention. The day-counting convention determines the way interest accrues over time for various financial instruments, such as bonds. The *actual/actual* convention means that we use the actual number of elapsed days and the actual number of days in a year, that is, 365 or 366. There are many other conventions such as *actual/365 (fixed)*, *actual/360*, and so on.

In *Step 3*, we selected two dates—valuation and expiry—as we are interested in pricing an option that expires in a year. It is important to set `ql.Settings.instance().evaluationDate` to the considered evaluation date to make sure the calculations are performed correctly. In this case, the dates only determine the passage of time, meaning that the option expires within a year. We would get the same results (with some margin of error due to the random component of the simulations) using different dates with the same interval between them.

We can check the time to expiry (in years) by running the following code:

```
T = day_counter.yearFraction(valuation_date, expiry_date)
print(f'Time to expiry in years: {T}')
```

Executing the snippet returns the following:

```
Time to expiry in years: 1.0
```

Next, we defined the option type (call/put), the type of exercise (European, American, or Bermudan), and the payoff (vanilla). In *Step 5*, we prepared the market data. We wrapped the values in quotes (`ql.SimpleQuote`) so that the values can be changed and those changes are properly registered in the instrument. This is an important step for calculating the Greeks in the *There's more...* section.

In *Step 6*, we defined the relevant curves. Simply put, `TARGET` is a calendar that contains information on which days are holidays.

In this step, we specified the three important components of the **Black-Scholes (BS)** process, which are:

- The price of the underlying instrument
- Volatility, which is constant as per our assumptions
- The risk-free rate, which is also constant over time

We passed all these objects to the Black-Scholes process (`ql.BlackScholesProcess`), which we defined in *Step 7*. Then, we passed the process object into the special engine used for pricing American options using Monte Carlo simulations (there are many predefined engines for different types of options and pricing methods). At this point, we provided the desired number of simulations, the number of time steps for discretization, and the degree/order of the polynomial in the LSMC algorithm. Additionally, we provided the random seed (`seedCalibration`) to make the results reproducible.

In *Step 9*, we created an instance of `ql.VanillaOption` by providing previously defined types of payoff and exercise. We also set the pricing engine to the one defined in *Step 8* using the `setPricingEngine` method.

Finally, we obtained the price of the option using the `NPV` method.

We can see that the option premium we obtained using QuantLib is very similar to the one we calculated previously, which further validates our results. The important thing to note here is that the workflow is similar for the valuation of a wide array of different derivatives, so it is good to be familiar with it. We could just as well price a European option using Monte Carlo simulations by substituting a few classes with their European option counterparts.



QuantLib also allows us to use variance reduction techniques such as antithetic values or control variates.

There's more...

Now that we have completed the preceding steps, we can calculate the Greeks. As we have mentioned in the previous recipe, the Greeks represent the sensitivity of the price of derivatives to a change in one of the underlying parameters (such as the price of the underlying asset, time to expiry, and so on).

When there is an analytical formula available for the Greeks (when the underlying QuantLib engine is using analytical formulas), we could just access it by running, for example, `option.delta()`. However, in cases such as valuations using binomial trees or simulations, there is no analytical formula, and we would receive an error (`RuntimeError: delta not provided`). This does not mean that it is impossible to calculate it, but we need to employ numerical differentiation and calculate it ourselves.

In this example, we will only extract the delta. Therefore, the relevant two-sided formula is:

$$\Delta = \frac{P(S_0 + h) - P(S_0 - h)}{2h}$$

Here, $P(S)$ is the price of the instrument given the underlying asset's price S ; h is a very small increment.

Run the following block of code to calculate the delta:

```
u_0 = u.value() # original value
h = 0.01

u.setValue(u_0 + h)
P_plus_h = option.NPV()

u.setValue(u_0 - h)
P_minus_h = option.NPV()

u.setValue(u_0) # set back to the original value

delta = (P_plus_h - P_minus_h) / (2 * h)
```

The simplest interpretation of the delta is that the option's delta equal to -1.36 indicates that, if the underlying stock increases in price by \$1 per share, the option on it will decrease by \$1.36 per share; otherwise, everything will be equal.

Pricing barrier options

A **barrier option** is a type of option that falls under the umbrella of exotic options. That is because they are more complex than plain European or American options. Barrier options are a type of path-dependent option because their payoff, and thus also their value, is based on the underlying asset's price path.

To be more precise, the payoff depends on whether or not the underlying asset has reached/exceeded a predetermined price threshold. Barrier options are typically classified as one of the following:

- A knock-out option, that is, the option becomes worthless if the underlying asset's price exceeds a certain threshold
- A knock-in option, that is, the option has no value until the underlying asset's price reaches a certain threshold

Considering the classes of the barrier options mentioned above, we can deal with the following categories:

- Up-and-Out: The option starts active and becomes worthless (knocked out) when the underlying asset's price moves up to the barrier level
- Up-and-In: The option starts inactive and becomes active (knocked in) when the underlying asset's price moves up to the barrier level
- Down-and-Out: The option starts active and becomes knocked out when the underlying asset's price moves down to the barrier level
- Down-and-In: The option starts inactive and becomes active when the underlying asset's price moves down to the barrier level

Other than the behavior described above, barrier options behave like standard call and put options.

In this recipe, we use Monte Carlo simulations to price an Up-and-In European call option with the underlying trading at \$55, a strike price of \$60, and a barrier level of \$65. The time to maturity will be 1 year.

How to do it...

Execute the following steps to price an Up-and-In European call option:

1. Import the libraries:

```
import numpy as np
from chapter_10_utils import simulate_gbm
```

2. Define the parameters for the valuation:

```
S_0 = 55
K = 60
BARRIER = 65
r = 0.06
sigma = 0.2
T = 1
N = 252
dt = T / N
N_SIMS = 10 ** 5
OPTION_TYPE = "call"
discount_factor = np.exp(-r * T)
```

3. Simulate the stock path using the `simulate_gbm` function:

```
gbm_sims = simulate_gbm(s_0=S_0, mu=r, sigma=sigma,
                         n_sims=N_SIMS, T=T, N=N)
```

4. Calculate the maximum value per path:

```
max_value_per_path = np.max(gbm_sims, axis=1)
```

5. Calculate the payoff:

```
payoff = np.where(max_value_per_path > BARRIER,
                   np.maximum(0, gbm_sims[:, -1] - K),
                   0)
```

6. Calculate the option's premium:

```
premium = discount_factor * np.mean(payoff)
premium
```

The premium of the considered Up-and-In European call option is 3.6267.

How it works...

In the first two steps, we imported the libraries (including the helper function, `simulate_gbm`, which we have already used throughout this chapter) and defined the parameters of the valuation.

In *Step 3*, we simulated 100,000 possible paths using a geometric Brownian motion. Then, we calculated the maximum price of the underlying asset for each path. Because we are working with an Up-and-In option, we just need to know if the maximum price of the asset reached the barrier level. If so, then the option's payoff at maturity will be equal to that of a vanilla European call. If the barrier level was not reached, the payoff from that path will be zero. We encoded that payoff condition in *Step 5*.

Lastly, we proceeded just as we have done with the European call option before—we took the average payoff and discounted it using the discount factor.



We can build some intuition about the prices of barrier options. For example, the price of an Up-and-Out barrier option should be lower than that of a vanilla equivalent. That is due to the fact that the payoffs of the two instruments would be identical except for the added risk that the Up-and-Out barrier option could be knocked-out before expiring. That added risk should be reflected in the lower price of such a barrier option as compared to its vanilla counterpart.

In this recipe, we have manually priced an Up-and-In European call option. However, we can also use the QuantLib library for the task. Due to the fact that there would be a lot of code repetition with the previous recipe, we do not show that in the book.

But you are highly encouraged to check out the solution using QuantLib in the accompanying notebook available on GitHub. We just mention that the solution using QuantLib returns the option premium of 3.6457, which is very close to the one we obtained manually. The difference can be attributed to the random component of the simulations.

There's more...

The valuation of barrier options is complex given those instruments are path-dependent. We have already mentioned how to use Monte Carlo simulations to price such options; however, there are several alternative approaches:

- Use a static replicating portfolio of vanilla options to mimic the value of the barrier at expiry and at a few discrete points in time along the barrier. Then, those options can be valued using the Black-Scholes model. By following this approach, we can obtain closed-form prices and replication strategies for all kinds of barrier options.
- Use the binomial tree approach to option pricing.
- Use the **partial differential equation (PDE)** and potentially combine it with the finite difference method.

Estimating Value-at-Risk using Monte Carlo

Value-at-Risk (VaR) is a very important financial metric that measures the risk associated with a position, portfolio, etc. It is commonly abbreviated to VaR, not to be confused with **vector autoregression** (which is abbreviated to **VAR**). VaR reports the worst expected loss—at a given level of confidence—over a certain horizon under normal market conditions. The easiest way to understand it is by looking at an example. Let's say that the 1-day 95% VaR of our portfolio is \$100. This means that 95% of the time (under normal market conditions), we will not lose more than \$100 by holding our portfolio over one day.



It is common to present the loss given by VaR as a positive (absolute) value. That is why in this example, a VaR of \$100 means losing no more than \$100. However, a negative VaR is possible and it would indicate a high probability of making a profit. For example, a 1-day 95% VaR of -\$100 would imply that our portfolio has a 95% chance of making more than \$100 over the next day.

There are several ways to calculate VaR, including:

- Parametric approach (variance-covariance)
- Historical simulation approach
- Monte Carlo simulations

In this recipe, we only consider the last method. We assume that we are holding a portfolio consisting of two assets (stocks of Intel and AMD) and that we want to calculate a 1-day Value-at-Risk.

How to do it...

Execute the following steps to estimate the Value-at-Risk using Monte Carlo simulations:

1. Import the libraries:

```
import numpy as np
import pandas as pd
import yfinance as yf
import seaborn as sns
```

2. Define the parameters that will be used for this recipe:

```
RISKY_ASSETS = ["AMD", "INTC"]
SHARES = [5, 5]
START_DATE = "2020-01-01"
END_DATE = "2020-12-31"
T = 1
N_SIMS = 10 ** 5
```

3. Download the price data from Yahoo Finance:

```
df = yf.download(RISKY_ASSETS, start=START_DATE,
                 end=END_DATE, adjusted=True)
```

4. Calculate the daily returns:

```
returns = df["Adj Close"].pct_change().dropna()
returns.plot(title="Intel's and AMD's daily stock returns in 2020")
```

Running the snippet results in the following figure.

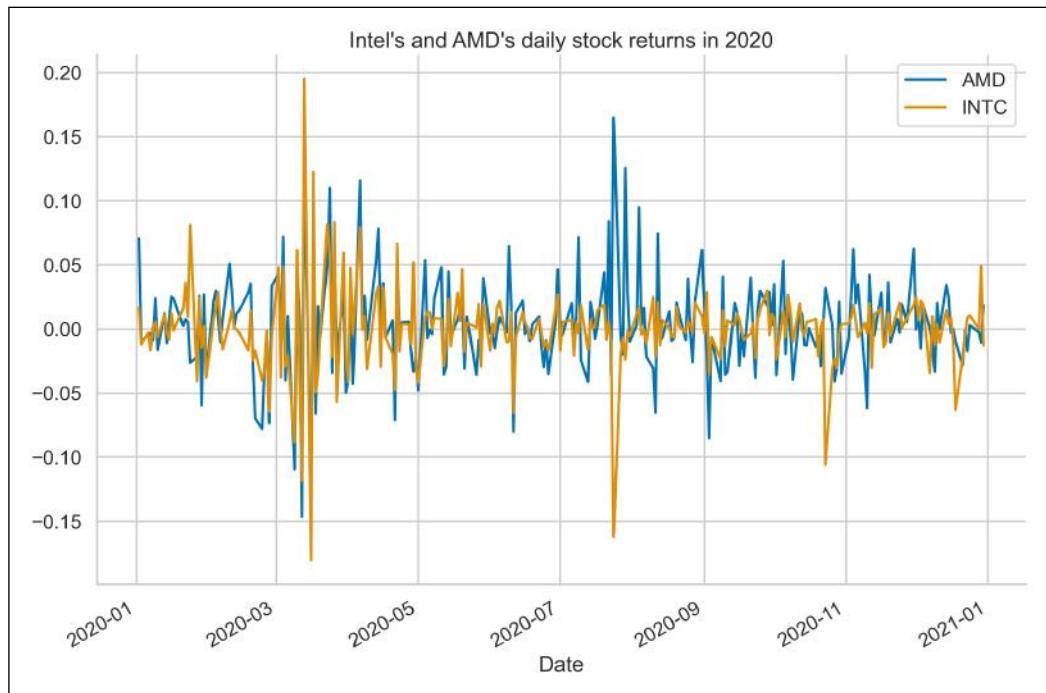


Figure 10.3: Simple returns of Intel and AMD in 2020

Additionally, we calculated the Pearson's correlation between the two series (using the `corr` method), which is equal to 0.5.

5. Calculate the covariance matrix:

```
cov_mat = returns.cov()
```

6. Perform the Cholesky decomposition of the covariance matrix:

```
chol_mat = np.linalg.cholesky(cov_mat)
```

7. Draw the correlated random numbers from the Standard Normal distribution:

```
rv = np.random.normal(size=(N_SIMS, len(RISKY_ASSETS)))
correlated_rv = np.transpose(
    np.matmul(chol_mat, np.transpose(rv)))
)
```

8. Define the metrics that will be used for simulations:

```
r = np.mean(returns, axis=0).values
sigma = np.std(returns, axis=0).values
S_0 = df["Adj Close"].values[-1, :]
P_0 = np.sum(SHARES * S_0)
```

9. Calculate the terminal price of the considered stocks:

```
S_T = S_0 * np.exp((r - 0.5 * sigma ** 2) * T +
                     sigma * np.sqrt(T) * correlated_rv)
```

10. Calculate the terminal portfolio value and the portfolio returns:

```
P_T = np.sum(SHARES * S_T, axis=1)
P_diff = P_T - P_0
```

11. Calculate the VaR for the selected confidence levels:

```
P_diff_sorted = np.sort(P_diff)
percentiles = [0.01, 0.1, 1.]
var = np.percentile(P_diff_sorted, percentiles)

for x, y in zip(percentiles, var):
    print(f'1-day VaR with {100-x}% confidence: ${-y:.2f}')
```

Running the snippet results in the following output:

```
1-day VaR with 99.99% confidence: $2.04
1-day VaR with 99.9% confidence: $1.48
1-day VaR with 99.0% confidence: $0.86
```

12. Present the results on a graph:

```
ax = sns.distplot(P_diff, kde=False)
ax.set_title("""Distribution of possible 1-day changes
in portfolio value 1-day 99% VaR""",
            fontsize=16)
ax.axvline(var[2], 0, 10000)
```

Running the snippet results in the following figure:

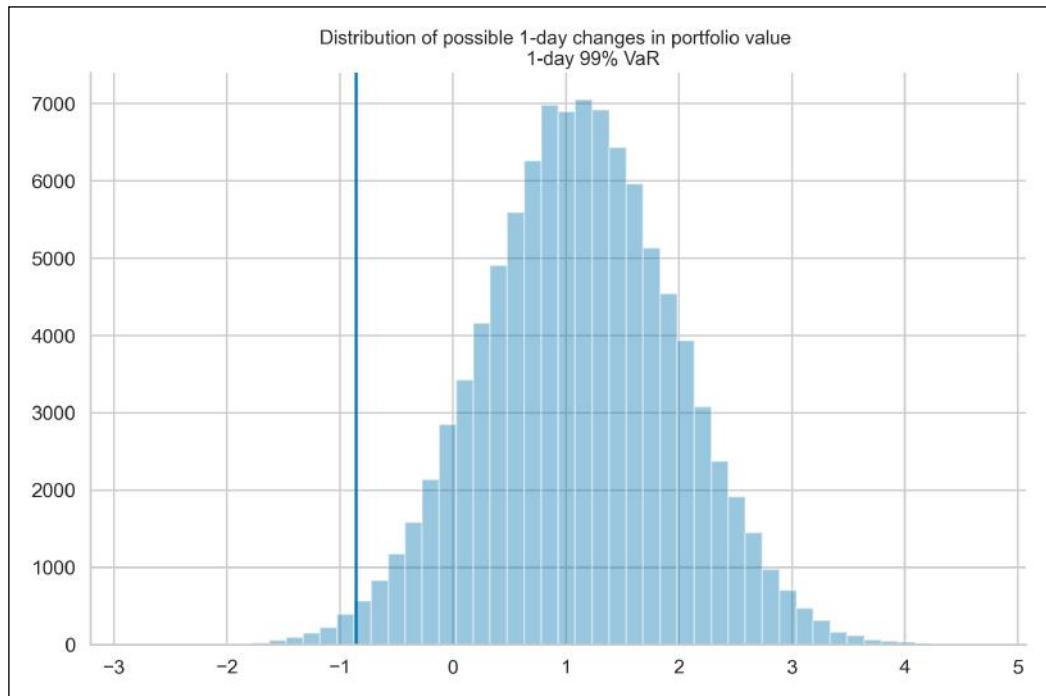


Figure 10.4: Distribution of the possible 1-day changes in portfolio value and the 1-day 99% VaR

Figure 10.4 shows the distribution of possible 1-day-ahead portfolio values. We present the 99% Value-at-Risk with the vertical line.

How it works...

In *Steps 2 to 4*, we downloaded the daily stock prices of Intel and AMD from the year 2020, extracted the adjusted close prices, and converted them into simple returns. We also defined a few parameters, such as the number of simulations and the number of shares we have in our portfolio.

There are two ways to approach VaR calculations:

- Calculate VaR from prices: Using the number of shares and the asset prices, we can calculate the worth of the portfolio now and its possible value X days ahead.
- Calculate VaR from returns: Using the percentage weights of each asset in the portfolio and the assets' expected returns, we can calculate the expected portfolio return X days ahead. Then, we can express VaR as the dollar amount based on that return and the current portfolio value.

The Monte Carlo approach to determining the price of an asset employs random variables drawn from the Standard Normal distribution. For the case of calculating portfolio VaR, we need to account for the fact that the assets in our portfolio may be correlated. To do so, in *Steps 5 to 7*, we generated correlated random variables. To do so, we first calculated the historical covariance matrix. Then, we used the Cholesky decomposition on it and multiplied the resulting matrix by the matrix containing the random variables.



Another possible approach to making random variables correlated is to use the **Singular Value Decomposition (SVD)** instead of the Cholesky decomposition. The function we can use for this is `np.linalg.svd`.

In *Step 8*, we calculated metrics such as the historical averages of the asset return, the accompanying standard deviations, the last known stock prices, and the initial portfolio value. In *Step 9*, we applied the analytical solution to the geometric Brownian motion SDE and calculated possible 1-day-ahead stock prices for both assets.

To calculate the portfolio VaR, we calculated the possible 1-day-ahead portfolio values and the accompanying differences ($P_T - P_0$). Then, we sorted them in ascending order. The $X\%$ VaR is simply the $(1-X)$ -th percentile of the sorted portfolio differences.



Banks frequently calculate the 1-day and 10-day VaR. To arrive at the latter, they can simulate the value of their assets over a 10-day interval using 1-day steps (discretization). However, they can also calculate the 1-day VaR and multiply it by the square root of 10. This might be beneficial for the bank if it leads to lower capital requirements.

There's more...

As we have mentioned, there are multiple ways of calculating the Value-at-Risk. And each of those comes with a set of potential drawbacks, some of which are:

- Assuming a parametric distribution (variance-covariance approach)

- Assuming that daily gains/losses are IID (independently and identically distributed)
- Not capturing enough tail risk
- Not considering the so-called Black Swan events (unless they are already in the historical sample)
- Historical VaR can be slow to adapt to new market conditions
- The historical simulation approach assumes that past returns are sufficient to evaluate future risk (connects to the previous points)



There are some interesting recent developments in using deep learning techniques, for example, generative adversarial networks for Value-at-Risk estimation.

Another general drawback of VaR is that it does not contain information about the size of the potential loss when it exceeds the threshold given by VaR. This is when **expected shortfall** (also known as conditional VaR or expected tail loss) comes into play. It simply states what the expected loss is in the worst X% of scenarios.

There are many ways to calculate the Expected Shortfall, but we present the one that is easily connected to the VaR and can be estimated using Monte Carlo simulations.

Following on from the example of a two-asset portfolio, we would like to know the following: if the loss exceeds the VaR, how big will it be? To obtain that number, we need to filter out all losses that are higher than the value given by VaR and calculate their expected value by taking the average.

We can do this using the following snippet:

```
var = np.percentile(P_diff_sorted, 5)
expected_shortfall = P_diff_sorted[P_diff_sorted<=var].mean()
```

Please bear in mind that for Expected Shortfall we only use a small fraction of all the simulations that were used to obtain the VaR. In *Figure 10.4*, we would only consider the observations to the left of the VaR line. That is why, in order to have reasonable results for the Expected Shortfall, the overall sample must be large enough.

The 1-day 95% VaR is \$0.29, while the accompanying Expected Shortfall is \$0.64. We can interpret these results as follows: if the loss exceeds the 95% VaR, we can expect to lose \$0.64 by holding our portfolio for 1 day.

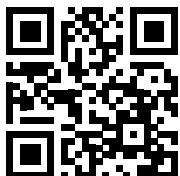
Summary

In this chapter, we have covered Monte Carlo simulations, which are a very versatile tool useful in many financial tasks. We demonstrated how to utilize them for simulating stock prices using a geometric Brownian motion, pricing various types of options (European, American, and Barrier), and calculating the Value-at-Risk.

However, in this chapter we have barely scratched the surface of all the possible applications of Monte Carlo simulations. In the following chapter, we also show how to use them to obtain the efficient frontier used for asset allocation.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

11

Asset Allocation

Asset allocation is the most important decision that any investor needs to face, and there is no one-size-fits-all solution that can work for each and every investor. By asset allocation, we mean spreading the investor's total investment amount over certain assets (be it stocks, options, bonds, or any other financial instruments). When considering the allocation, the investor wants to balance the risk and the potential reward. At the same time, the allocation is dependent on factors such as the individual goals (expected return), risk tolerance (how much risk the investor is willing to accept), or the investment horizon (short-or long-term investment).

The key framework in asset allocation is the **modern portfolio theory** (MPT, also known as **mean-variance analysis**). It was introduced by the Nobel recipient Harry Markowitz and describes how risk-averse investors can construct portfolios to maximize their expected returns (profits) for a given level of risk. The main insight from MPT is that investors should not evaluate an asset's performance alone (by metrics such as expected return or volatility), but instead, investigate how it would impact the performance of their portfolio of assets.

MPT is closely related to the concept of diversification, which simply means that owning different kinds of assets reduces risk, as the loss or gain of a particular security has less impact on the overall portfolio's performance. Another key concept to be aware of is that while the portfolio return is the weighted average of the individual asset returns, this is not true for the risk (volatility). That is because the volatility is also dependent on the correlations between the assets. What is interesting is that thanks to optimized asset allocation, it is possible to have a portfolio with lower volatility than the lowest individual volatility of the assets in the portfolio. In principle, the lower the correlation between the assets we hold, the better it is for diversification. With a perfect negative correlation, we could diversify all the risk.

The main assumptions of modern portfolio theory are:

- Investors are rational and aim to maximize their returns while avoiding risks whenever possible.
- Investors share the goal to maximize their expected returns.
- All investors have the same level of information about potential investments.
- Commissions, taxes, and transaction costs are not taken into account.
- Investors can borrow and lend money (without limits) at a risk-free rate.

In this chapter, we start with the most basic asset allocation strategy, and on its basis, learn how to evaluate the performance of portfolios (also applicable to individual assets). Later on, we show three different approaches to obtaining the efficient frontier, while also relaxing some of the assumptions of MPT. One of the main benefits of learning how to approach optimization problems is that they can be easily refactored, for example, by optimizing a different objective function or adding specific constraints on the weights. This requires only slight modifications to the code, while the majority of the framework stays the same. At the very end, we explore a novel approach to asset allocation based on the combination of graph theory and machine learning—Hierarchical Risk Parity.

We cover the following recipes in this chapter:

- Evaluating an equally-weighted portfolio's performance
- Finding the efficient frontier using Monte Carlo simulations
- Finding the efficient frontier using optimization with SciPy
- Finding the efficient frontier using convex optimization with CVXPY
- Finding the optimal portfolio with Hierarchical Risk Parity

Evaluating an equally-weighted portfolio's performance

We begin with inspecting the most basic asset allocation strategy: the **equally-weighted ($1/n$) portfolio**. The idea is to assign equal weights to all the considered assets, thus diversifying the portfolio. As simple as that might sound, DeMiguel, Garlappi, and Uppal (2007) show that it can be difficult to beat the performance of the $1/n$ portfolio by using more advanced asset allocation strategies.

The goal of the recipe is to show how to create a $1/n$ portfolio of the FAANG companies (Facebook/Meta, Amazon, Apple, Netflix, and Google/Alphabet), calculate its returns, and then use the `quantstats` library to quickly obtain all relevant portfolio evaluation metrics in the form of a tear sheet. Historically, a tear sheet is a concise (usually one-page) document summarizing important information about public companies.

How to do it...

Execute the following steps to create and evaluate the $1/n$ portfolio:

1. Import the libraries:

```
import yfinance as yf
import numpy as np
import pandas as pd
import quantstats as qs
```

2. Define the considered assets and download their prices from Yahoo Finance:

```
ASSETS = ["META", "AMZN", "AAPL", "NFLX", "GOOG"]
n_assets = len(ASSETS)

prices_df = yf.download(ASSETS,
                       start="2020-01-01",
                       end="2021-12-31",
                       adjusted=True)
```

3. Calculate individual asset returns:

```
returns = prices_df["Adj Close"].pct_change().dropna()
```

4. Define the weights:

```
portfolio_weights = n_assets * [1 / n_assets]
```

5. Calculate the portfolio returns:

```
portfolio_returns = pd.Series(
    np.dot(portfolio_weights, returns.T),
    index=returns.index
)
```

6. Generate basic performance evaluation plots:

```
qs.plots.snapshot(portfolio_returns,
                   title="1/n portfolio's performance",
                   grayscale=True)
```

Executing the snippet generates the following figure:

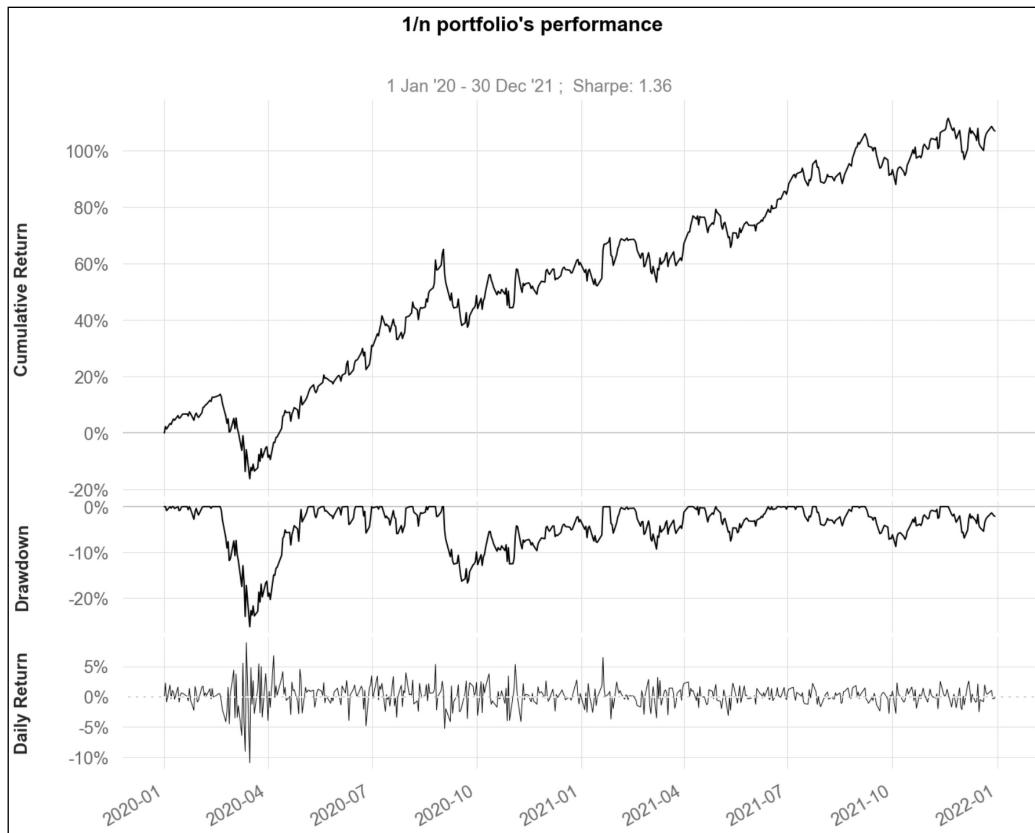


Figure 11.1: Selected evaluation metrics of the 1/n portfolio

The created snapshot consists of cumulative portfolio returns, the underwater plot depicting the drawdown periods (we will explain it in the *How it works...* section), and daily returns.

7. Calculate the basic portfolio evaluation metrics:

```
qs.reports.metrics(portfolio_returns,
                    benchmark="SPY",
                    mode="basic",
                    prepare_returns=False)
```

Executing the snippet returns the following metrics for our portfolio and the benchmark:

	Strategy	Benchmark
Start Period	2020-01-02	2020-01-02
End Period	2021-12-30	2021-12-30
Risk-Free Rate	0.0%	0.0%
Time in Market	100.0%	100.0%
Cumulative Return	107.0%	51.3%
CAGR%	44.02%	23.07%
Sharpe	1.36	0.94
Sortino	1.96	1.3
Sortino/V2	1.39	0.92
Omega	1.28	1.28
Max Drawdown	-26.35%	-33.72%
Longest DD Days	140	172
Gain/Pain Ratio	0.28	0.22
Gain/Pain (1M)	3.48	1.54
Payoff Ratio	0.84	0.8
Profit Factor	1.28	1.22
Common Sense Ratio	1.21	0.97
CPC Index	0.63	0.57
Tail Ratio	0.94	0.8
Outlier Win Ratio	4.13	5.68
Outlier Loss Ratio	3.3	4.46
MTD	1.56%	4.89%
3M	8.24%	10.17%
6M	11.59%	12.08%
YTD	29.09%	29.05%
1Y	28.21%	29.89%
3Y (ann.)	44.02%	23.07%
5Y (ann.)	44.02%	23.07%
10Y (ann.)	44.02%	23.07%
All-time (ann.)	44.02%	23.07%
Avg. Drawdown	-3.55%	-1.78%
Avg. Drawdown Days	18	10
Recovery Factor	4.06	1.52
Ulcer Index	0.06	0.07
Serenity Index	2.54	0.8

Figure 11.2: Performance evaluation metrics of the $1/n$ portfolio and the S&P 500 benchmark

We describe some of the metrics presented in Figure 11.2 in the following section.

How it works...

In Steps 1 to 3, we followed the already established approach—imported the libraries, set up the parameters, downloaded stock prices of the FAANG companies from the years 2020 to 2021, and calculated simple returns using the adjusted close prices.

In Step 4, we created a list of weights, each one equal to $1/n_{\text{assets}}$, where n_{assets} is the number of assets we want to have in our portfolio. Next, we calculated the portfolio returns as a matrix multiplication (also known as the dot product) of the portfolio weights and a transposed matrix of asset returns. To transpose the matrix, we used the `T` method of a pandas DataFrame. Then, we stored the portfolio returns as a pandas Series object, because that is the input for the ensuing step.



In the first edition of the book, we explored the performance of the $1/n$ portfolio using the `pyfolio` library. However, since that time, the company that was responsible for the library (Quantopian) was closed, and the library is not actively maintained anymore. The library can still be used, as we show in the additional notebook available in the book's GitHub repository. Alternatively, you can use `pyfolio-reloaded`, which is a fork of the original library maintained by Stefan Jansen, the author of *Machine Learning for Algorithmic Trading*.

In Step 6, we generated a figure containing basic portfolio evaluation plots using the `quantstats` library. While we are already familiar with the plot depicting the daily returns, the other two are new:

- **Cumulative returns plot:** It presents the evolution of the portfolio's worth over time.
- **Underwater plot:** This plot presents the investment from a pessimistic point of view, as it focuses on losses. It plots all the drawdown periods and how long they lasted, that is, until the value rebounded to a new high. One of the insights we can draw from this is how long the periods of losses lasted.

Lastly, we generated portfolio evaluation metrics. While doing so, we also provided a benchmark. We chose the SPY, which is an **exchange-traded fund (ETF)** designed to follow the S&P 500 index. We could provide the benchmark as either the ticker or a pandas DataFrame/Series containing the prices/returns. The library can handle both options and we can indicate if we want to calculate the returns from prices using the `prepare_returns` argument.

The most important metrics that we saw in *Figure 11.2* are:

- **Sharpe ratio:** One of the most popular performance evaluation metrics, it measures the excess return (over the risk-free rate) per unit of standard deviation. When no risk-free rate is provided, the default assumption is that it is equal to 0%. The greater the Sharpe ratio, the better the portfolio's risk-adjusted performance.
- **Sortino ratio:** A modified version of the Sharpe ratio, where the standard deviation in the denominator is replaced with downside deviation.
- **Omega ratio:** The probability-weighted ratio of gains over losses for a determined return target threshold (default set to 0). Its main advantage over the Sharpe ratio is that the Omega ratio—by construction—considers all moments of the returns distribution, while the former only considers the first two (mean and variance).
- **Max drawdown:** A metric of the downside risk of a portfolio, it measures the largest peak-to-valley loss (expressed as a percentage) during the course of the investment. The lower the maximum drawdown, the better.

- **Tail ratio:** The ratio (absolute) between the 95th and 5th percentile of the daily returns. A tail ratio of ~0.8 means that losses are ~1.25 times as bad as profits.



Downside deviation is similar to standard deviation; however, it only considers negative returns—it discards all positive changes from the series. It also allows us to define different levels of minimum acceptable returns (dependent on the investor) and returns below that threshold are used to calculate the downside deviation.

There's more...

So far, we have mostly generated only the basic selection of plots and metrics available in the `quantstats` library. However, the library has much more to offer.

Full tear sheets

`quantstats` allows us to generate a complete HTML report containing all of the available plots and metrics (including a comparison to the benchmark). We can create such a report using the following command:

```
qs.reports.html(portfolio_returns,  
                benchmark="SPY",  
                title="1/n portfolio",  
                download_filename="EW portfolio evaluation.html")
```

Executing it generates an HTML file containing the exhaustive tear sheet of our equally-weighted portfolio, compared to the SPY. Please refer to the `EW portfolio evaluation.html` file on GitHub.

First, let's explain some of the new, yet relevant metrics visible in the generated report:

- **Calmar ratio:** The ratio is defined as the average annual compounded rate of return divided by the maximum drawdown for that same time period. The higher the ratio, the better.
- **Skew:** Skewness measures the degree of asymmetry, that is, how much is the given distribution (here, of portfolio returns) more skewed than the Normal distribution. Negative skewness (left-skewed distributions) means that large negative returns occur more frequently than large positive ones.
- **Kurtosis:** It measures extreme values in either of the tails. Distributions with large kurtosis exhibit tail data exceeding the tails of the Gaussian distribution, meaning that large and small returns occur more frequently.
- **Alpha:** It describes a strategy's ability to beat the market. In other words, it is the portfolio excess returns above the benchmark return.
- **Beta:** It measures the overall systematic risk of a portfolio of investments. In other words, it is a measure of portfolio volatility compared to the systematic risk of the entire market. A portfolio's beta is equal to the weighted average of the beta coefficients of all the individual assets in a portfolio.

The metrics also include the 10 worst drawdowns. That is, they show how bad each of the drawdowns was, the recovery date, and the drawdowns' duration. This information complements the analysis of the underwater plot we mentioned before.

Worst 10 Drawdowns			
Started	Recovered	Drawdown	Days
2020-02-20	2020-05-07	-26.35%	77
2020-09-03	2021-01-21	-16.75%	140
2021-01-27	2021-04-05	-9.33%	68
2021-09-08	2021-11-12	-8.75%	65
2021-04-30	2021-06-14	-7.55%	45
2021-11-22	2021-12-30	-6.91%	38
2020-07-13	2020-08-04	-5.93%	22
2020-06-24	2020-07-01	-5.81%	7
2020-08-07	2020-08-18	-4.27%	11
2021-07-27	2021-08-27	-4.21%	31

Figure 11.3: The 10 worst drawdowns during the evaluation period

Then, the report also contains some new plots, which we explain below:

- **Rolling Sharpe ratio:** Instead of reporting one number over time, it is also interesting to see how stable the Sharpe ratio was. That is why the following plot presents this metric calculated on a rolling basis, using 6 months' worth of data.

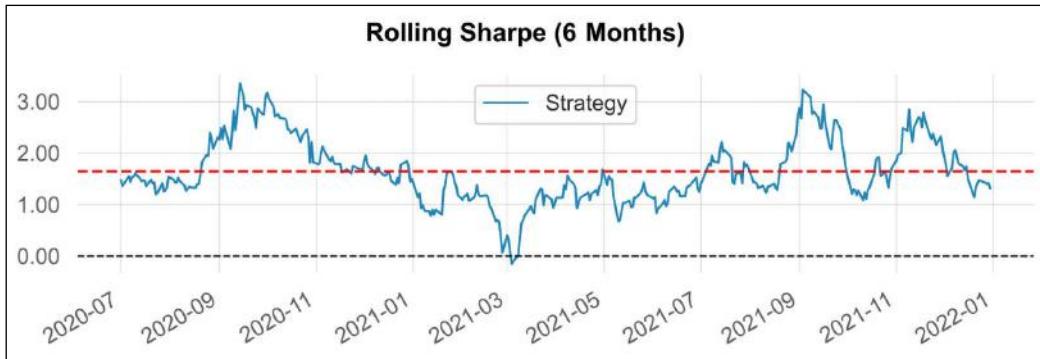


Figure 11.4: Rolling (6 months) Sharpe ratio

- The five worst drawdown periods are also visualized on a separate plot. For exact dates when the drawdowns started and ended, please refer to *Figure 11.3*. One thing worth mentioning is that the drawdown periods are superimposed on the cumulative returns plot. This way, we can clearly confirm the definition of the drawdown, that is, how much our portfolio is down from the peak before it recovers back to the peak level.

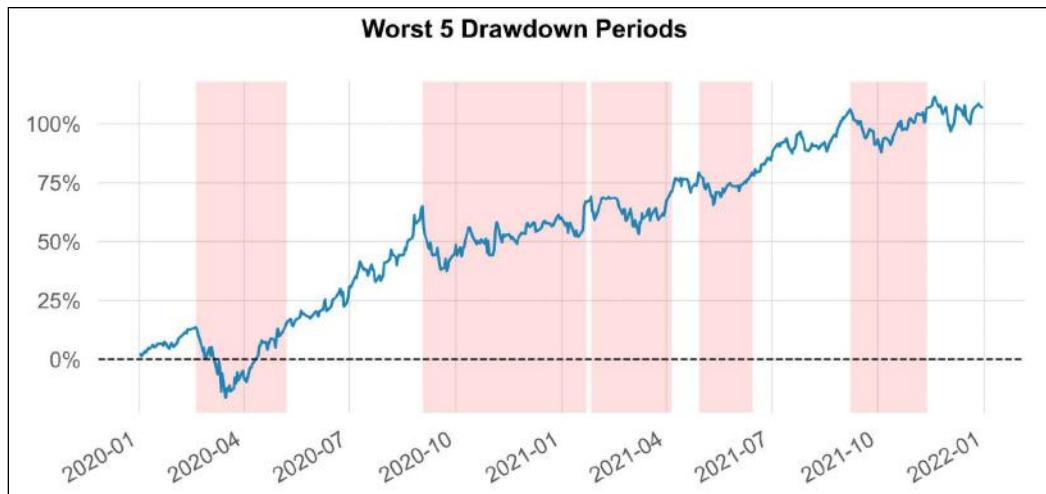


Figure 11.5: Five worst drawdown periods during the evaluation period

- A histogram depicting the distribution of the monthly returns, including a kernel density estimate (KDE) and the average value. It's helpful in analyzing the distribution of the returns. In the plot, we can see that the average monthly returns over the evaluation period were positive.

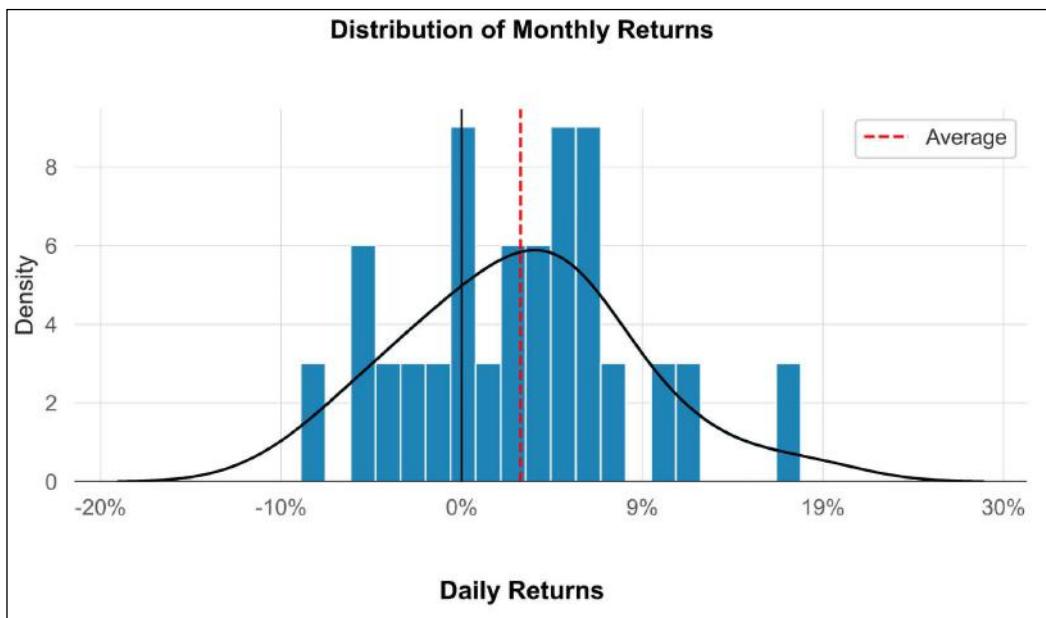


Figure 11.6: Distribution of the monthly returns (histogram + KDE)

- A heatmap serving as a summary of what the returns were over certain months/years.

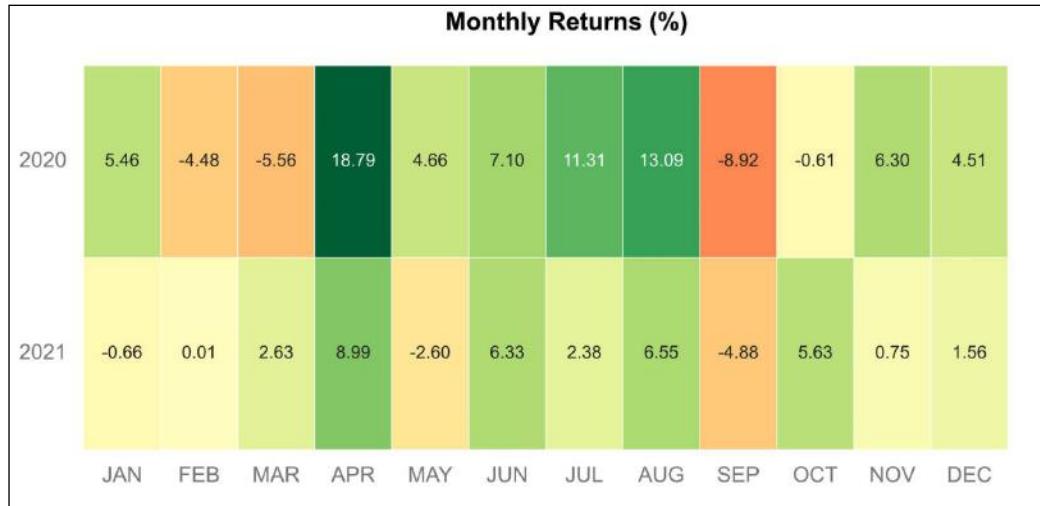


Figure 11.7: A heatmap presenting the monthly returns over the years

- A quantile plot showing the distribution of the returns aggregated to different frequencies.

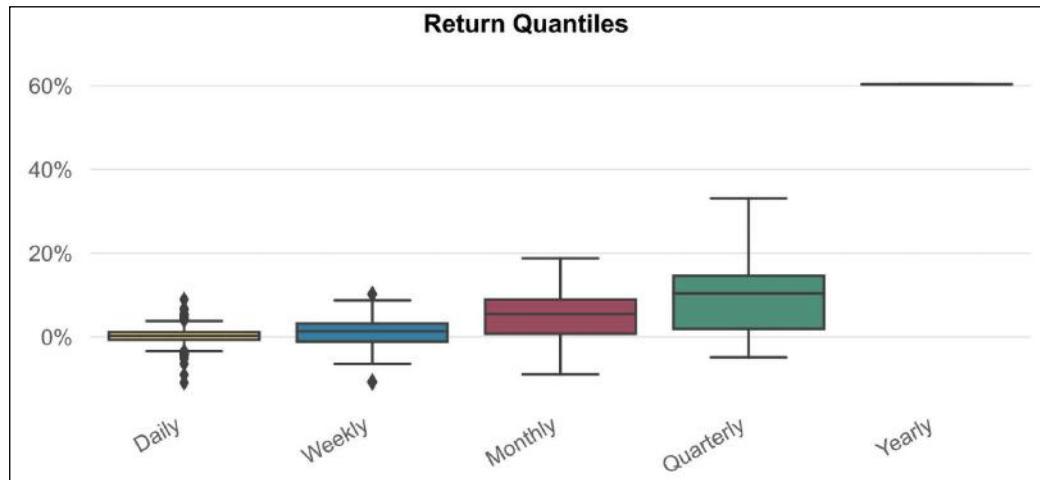


Figure 11.8: Quantile plot aggregating the returns to different frequencies

Before creating the comprehensive HTML report, we generated the basic plots and metrics using the `qs.reports.plots` and `qs.reports.metrics` functions. We can also use those functions to get the very same metrics/plots as we have obtained in the report by appropriately specifying the `mode` argument. To get all the metrics, we should pass "full" instead of "basic" (which is also the default value).

Enriching the pandas DataFrames/Series with new methods

Another interesting feature of the `quantstats` library is that it can enrich the pandas DataFrame or Series with new methods, used for calculating all the metrics available in the library. To do so, we first need to execute the following command:

```
qs.extend_pandas()
```

Then, we can access the methods straight from the DataFrame containing the return series. For example, we can quickly calculate the Sharpe and Sortino ratios using the following snippet:

```
print(f"Sharpe ratio: {portfolio_returns.sharpe():.2f}")
print(f"Sortino ratio: {portfolio_returns.sortino():.2f}")
```

Which returns:

```
Sharpe ratio: 1.36
Sortino ratio: 1.96
```

The values are a match to what we calculated earlier using the `qs.reports.metrics` function. For a complete list of the available methods, you can run the following snippet:

```
[method for method in dir(qs.stats) if method[0] != "_"]
```

See also

Additional resources are available here:

- DeMiguel, V., Garlappi, L., & Uppal, R. 2007, “Optimal versus naive diversification: how inefficient is the 1/N portfolio strategy?” *The Review of Financial Studies*, 22(5): 1915-1953: <https://doi.org/10.1093/rfs/hhm075>

Finding the efficient frontier using Monte Carlo simulations

According to the Modern Portfolio Theory, the **efficient frontier** is a set of optimal portfolios in the risk-return spectrum. This means that the portfolios on the frontier:

- Offer the highest expected return for a given level of risk
- Offer the lowest level of risk for a given level of expected returns

All portfolios located under the efficient frontier curve are considered sub-optimal, so it is always better to choose the ones on the frontier instead.

In this recipe, we show how to find the efficient frontier using Monte Carlo simulations. Before showing more elegant approaches based on optimization, we employ a brute force approach in which we build thousands of portfolios using randomly assigned weights. Then, we can calculate the portfolios’ performance (expected returns/volatility) and use those values to determine the efficient frontier. For this exercise, we use the returns of four US tech companies from 2021.

How to do it...

Execute the following steps to find the efficient frontier using Monte Carlo simulations:

1. Import the libraries:

```
import yfinance as yf
import numpy as np
import pandas as pd
```

2. Set up the parameters:

```
N_PORTFOLIOS = 10 ** 5
N_DAYS = 252
ASSETS = ["META", "TSLA", "TWTR", "MSFT"]
ASSETS.sort()

n_assets = len(ASSETS)
```

3. Download the stock prices from Yahoo Finance:

```
prices_df = yf.download(ASSETS,
                       start="2021-01-01",
                       end="2021-12-31",
                       adjusted=True)
```

4. Calculate the annualized average returns and the corresponding standard deviation:

```
returns_df = prices_df["Adj Close"].pct_change().dropna()
avg_returns = returns_df.mean() * N_DAYS
cov_mat = returns_df.cov() * N_DAYS
```

5. Simulate random portfolio weights:

```
np.random.seed(42)
weights = np.random.random(size=(N_PORTFOLIOS, n_assets))
weights /= np.sum(weights, axis=1)[:, np.newaxis]
```

6. Calculate the portfolio metrics:

```
portf_rtns = np.dot(weights, avg_returns)

portf_vol = []
for i in range(0, len(weights)):
    vol = np.sqrt(
        np.dot(weights[i].T, np.dot(cov_mat, weights[i])))
)
portf_vol.append(vol)
```

```

portf_vol = np.array(portf_vol)

portf_sharpe_ratio = portf_rtns / portf_vol

```

7. Create a DataFrame containing all the data:

```

portf_results_df = pd.DataFrame(
    {"returns": portf_rtns,
     "volatility": portf_vol,
     "sharpe_ratio": portf_sharpe_ratio}
)

```

The DataFrame looks as follows:

	returns	volatility	sharpe_ratio
0	0.335464	0.266351	1.259480
1	0.049227	0.346262	0.142167
2	0.175372	0.267961	0.654471
3	0.291582	0.257783	1.131113
4	0.346867	0.264669	1.310569
...
99995	0.442953	0.279346	1.585678
99996	0.145918	0.304849	0.478655
99997	0.419965	0.278450	1.508224
99998	0.277796	0.288633	0.962455
99999	0.353457	0.305080	1.158572

Figure 11.9: Selected metrics of each of the generated portfolios

8. Locate the points creating the efficient frontier:

```

N_POINTS = 100

ef rtn_list = []
ef vol_list = []

possible_ef_rtns = np.linspace(portf_results_df["returns"].min(),
                                portf_results_df["returns"].max(),
                                N_POINTS)
possible_ef_rtns = np.round(possible_ef_rtns, 2)
portf_rtns = np.round(portf_rtns, 2)

```

```
for rtn in possible_ef_rtns:
    if rtn in portf_rtns:
        ef_rtn_list.append(rtn)
        matched_ind = np.where(portf_rtns == rtn)
        ef_vol_list.append(np.min(portf_vol[matched_ind]))
```

9. Plot the efficient frontier:

```
MARKERS = ["o", "X", "d", "*"]

fig, ax = plt.subplots()
portf_results_df.plot(kind="scatter", x="volatility",
                      y="returns", c="sharpe_ratio",
                      cmap="RdYlGn", edgecolors="black",
                      ax=ax)
ax.set(xlabel="Volatility",
       ylabel="Expected Returns",
       title="Efficient Frontier")
ax.plot(ef_vol_list, ef_rtn_list, "b--")
for asset_index in range(n_assets):
    ax.scatter(x=np.sqrt(cov_mat.iloc[asset_index, asset_index]),
               y=avg_returns[asset_index],
               marker=MARKERS[asset_index],
               s=150, color="black",
               label=ASSETS[asset_index])
ax.legend()
plt.show()
```

Executing the snippet generates the plot with all the randomly created portfolios, four points indicating the individual assets, and the efficient frontier.

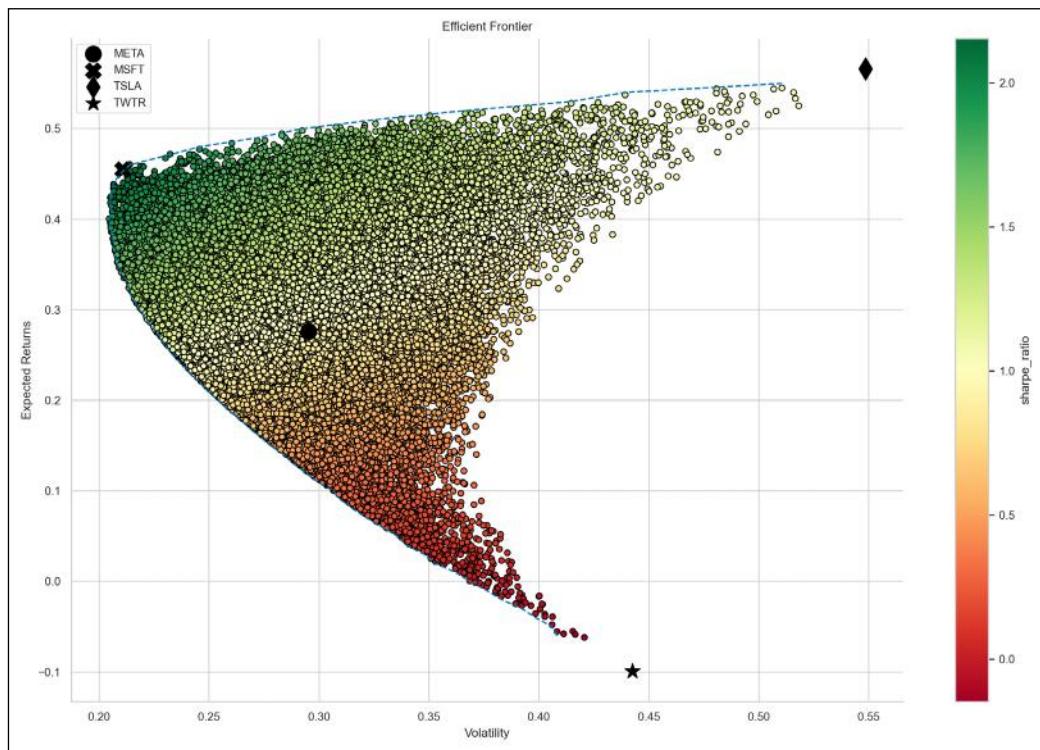


Figure 11.10: The efficient frontier identified using Monte Carlo simulations

In *Figure 11.10*, we see the typical, bullet-like shape of the efficient frontier.

Some insights we could draw from analyzing the efficient frontier:

- Anything to the left of the efficient frontier's line is not achievable, as we cannot get that level of expected return for such a level of volatility.
- The performance of a portfolio consisting only of Microsoft's stock lies very close to the efficient frontier.

Ideally, we should search for a portfolio offering exceptional returns but with a combined standard deviation that is lower than the standard deviations of the individual assets. For example, we should not consider a portfolio consisting only of Meta's stock (it is not efficient), but the one that lies on the frontier directly above. That is because the latter offers a much better expected return for the same level of expected volatility.

How it works...

In *Step 2*, we defined the parameters used for this recipe, such as the considered timeframe, the assets we wanted to use for building the portfolio, and the number of simulations. An important thing to note here is that we also ran `ASSETS.sort()` to sort the list alphabetically. This matters when interpreting the results, as when downloading data from Yahoo Finance using the `yfinance` library, the obtained prices are ordered alphabetically, not as specified in the provided list. Having downloaded the stock prices, we calculated simple returns using the `pct_change` method, and dropped the first row containing NaNs.

For evaluating the potential portfolios, we needed the average (expected) annual return and the corresponding covariance matrix. We obtained them by using the `mean` and `cov` methods of the `DataFrame`. We also annualized both metrics by multiplying them by 252 (the average number of trading days in a year).



We needed the covariance matrix, as for calculating the portfolio volatility, we also needed to account for the correlation between the assets. To benefit from significant diversification, the assets should have low positive or negative correlations.

In *Step 5*, we calculated the random portfolio weights. Following the assumptions of the modern portfolio theory (refer to the chapter introduction for reference), the weights needed to be positive and sum up to 1. To achieve this, we first generated a matrix of random numbers (between 0 and 1) using `np.random.random`. The matrix was of size `N_SIMULATIONS` by `n_assets`. To make sure the weights summed up to 1, we divided each row of the matrix by its sum.

In *Step 6*, we calculated the portfolio metrics—returns, standard deviation, and the Sharpe ratio. To calculate the expected annual portfolio returns, we had to multiply the weights by the previously calculated annual averages. For the standard deviations, we had to use the following formula: $\omega^T \Sigma \omega$, where ω is the vector of weights and Σ is the historical covariance matrix. We iterated over all the simulated portfolios using a `for` loop.



In this case, the `for` loop implementation is actually faster than the vectorized matrix equivalent: `np.diag(np.sqrt(np.dot(weights, np.dot(cov_mat, weights.T)))))`. The reason for that is the quickly increasing number of off-diagonal elements to be calculated, which does not matter for the metrics of interest. This approach is faster than the `for` loop for only a relatively small number of simulations (~100).

For this example, we assumed that the risk-free rate was 0%, so the Sharpe ratio of the portfolio could be calculated as portfolio returns divided by the portfolio's volatility. Another possible approach would be to calculate the average annual risk-free rate over 2021 and to use the portfolio excess returns for calculating the ratio.



One thing to keep in mind while finding the optimal asset allocation and evaluating its performance is that we are optimizing historically. We use the past performance to select the allocation that should work best, provided the market conditions do not change. As we know very well, that is rarely the case, thus past performance is not always indicative of future performance.

The last three steps led to visualizing the results. First, we put all the relevant metrics into a pandas DataFrame. Second, we identified the points of the efficient frontier. To do so, we created an array of expected returns from the sample. We used `np.linspace`, with the min and max values coming from the calculated portfolio returns. We rounded the numbers to two decimals to make the calculations smoother. For each expected return, we found the minimum observed volatility. In cases where there was no match, as can happen with equally spread points on the linear space, we skipped that point.

In the very last step, we plotted the simulated portfolios, the individual assets, and the approximated efficient frontier in one plot. The shape of the frontier was a bit jagged, which can be expected when using only simulated values that are not that frequent in some extreme areas. Additionally, we colored the dots representing the simulated portfolios by the value of the Sharpe ratio. Following the ratio's definition, the upper-left part of the plot shows a sweet spot with the highest expected returns per expected volatility.



You can find the available colormaps in `matplotlib` documentation. Depending on the problem at hand, a different colormap might be more suitable (sequential, diverging, qualitative, and so on).

There's more...

Having simulated 100,000 random portfolios, we can also investigate which one has the highest Sharpe ratio (maximum expected return per unit of risk, also known as the **tangency portfolio**) or minimum volatility. To locate these portfolios among the simulated ones, we use the `np.argmax` and `np.argmax` functions, which return the index of the minimum/maximum value in the array.

The code is as follows:

```
max_sharpe_ind = np.argmax(portf_results_df["sharpe_ratio"])
max_sharpe_portf = portf_results_df.loc[max_sharpe_ind]

min_vol_ind = np.argmin(portf_results_df["volatility"])
min_vol_portf = portf_results_df.loc[min_vol_ind]
```

We can also investigate the constituents of these portfolios, together with the expected performance. Here, we only focus on the results, but the code used for generating the summaries is available in the book's GitHub repository.

The maximum Sharpe ratio portfolio allocates the majority of the resources (~95%) to Microsoft and virtually nothing to Twitter. That is because Twitter's annualized average returns for 2021 were negative:

```
Maximum Sharpe Ratio portfolio ----
Performance
returns: 45.14% volatility: 20.95% sharpe_ratio: 215.46%
Weights
META: 2.60% MSFT: 95.17% TSLA: 2.04% TWTR: 0.19%
```

The minimum volatility portfolio assigns ~78% of the weight to Microsoft, as it is the stock with the lowest volatility (this can be inspected by viewing the covariance matrix):

```
Minimum Volatility portfolio ----
Performance
returns: 40.05% volatility: 20.46% sharpe_ratio: 195.76%
Weights
META: 17.35% MSFT: 78.16% TSLA: 0.23% TWTR: 4.26%
```

Lastly, we mark these two portfolios on the efficient frontier plot. To do so, we add two extra scatterplots, each with one point corresponding to the selected portfolio. We then define the marker shape with the `marker` argument and the marker size with the `s` argument. We increase the size of the markers to make the portfolios more visible among all other points.

The code is as follows:

```
fig, ax = plt.subplots()
portf_results_df.plot(kind="scatter", x="volatility",
                      y="returns", c="sharpe_ratio",
                      cmap="RdYlGn", edgecolors="black",
                      ax=ax)
ax.scatter(x=max_sharpe_portf["volatility"],
            y=max_sharpe_portf["returns"],
            c="black", marker="*",
            s=200, label="Max Sharpe Ratio")
ax.scatter(x=min_vol_portf["volatility"],
            y=min_vol_portf["returns"],
            c="black", marker="P",
            s=200, label="Minimum Volatility")
ax.set(xlabel="Volatility", ylabel="Expected Returns",
       title="Efficient Frontier")
ax.legend()
plt.show()
```

Executing the snippet generates the following figure:

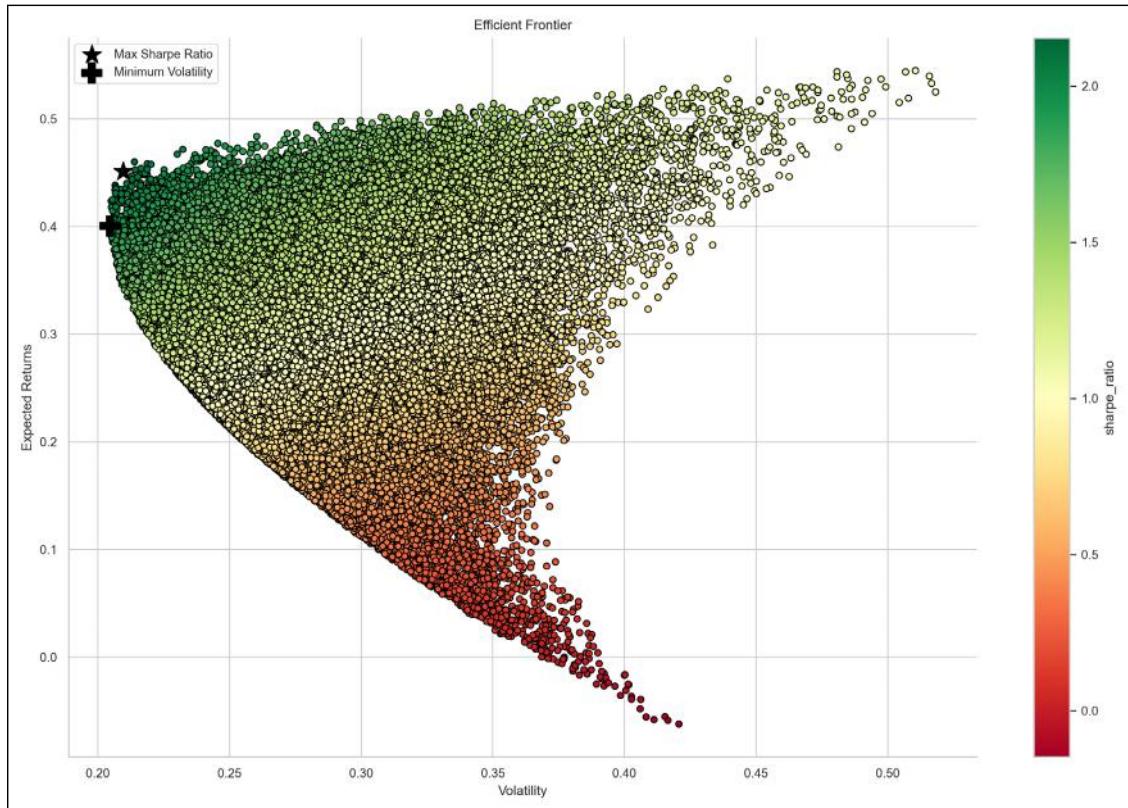


Figure 11.11: Efficient frontier with the Global Minimum Volatility and Max Sharpe Ratio portfolios

We did not plot the individual assets and the efficient frontier's line to avoid the plot becoming too cluttered. The plot aligns with the intuition we have built while analyzing *Figure 11.10*. First, the Minimum Volatility portfolio lies on the leftmost part of the frontier, which corresponds to the lowest expected volatility. Second, the Max Sharpe Ratio portfolio lies in the upper-left part of the plot, where the ratio of the expected returns to volatility is the highest.

Finding the efficient frontier using optimization with SciPy

In the previous recipe, *Finding the efficient frontier using Monte Carlo simulations*, we used a brute force approach based on Monte Carlo simulations to visualize the efficient frontier. In this recipe, we use a more refined method to find the frontier.

From its definition, the efficient frontier is formed by a set of portfolios offering the highest expected portfolio return for certain volatility, or offering the lowest risk (volatility) for a certain level of expected returns. We can leverage this fact, and use it in numerical optimization.

The goal of optimization is to find the best (optimal) value of the objective function by adjusting the target variables and taking into account some boundaries and constraints (which have an impact on the target variables). In this case, the objective function is a function returning portfolio volatility, and the target variables are portfolio weights.

Mathematically, the problem can be expressed as:

$$\min \omega^T \Sigma \omega$$

$$s.t \quad \omega^T \mathbf{1} = 1$$

$$\omega \geq 0$$

$$\omega^T \mu = \mu_p$$

Here, ω is a vector of weights, Σ is the covariance matrix, μ is a vector of returns, and μ_p is the expected portfolio return.

To find the efficient frontier, we iterate the optimization routine used for finding the optimal portfolio weights over a range of expected portfolio returns.

In this recipe, we work with the same dataset as in the previous one in order to show that the results obtained by both approaches are similar.

Getting ready

This recipe requires running all the code from the *Finding the efficient frontier using Monte Carlo simulations* recipe.

How to do it...

Execute the following steps to find the efficient frontier using optimization with SciPy:

1. Import the libraries:

```
import numpy as np
import scipy.optimize as sco
from chapter_11_utils import print_portfolio_summary
```

2. Define functions for calculating portfolio returns and volatility:

```
def get_portf_rtn(w, avg_rtns):
    return np.sum(avg_rtns * w)

def get_portf_vol(w, avg_rtns, cov_mat):
    return np.sqrt(np.dot(w.T, np.dot(cov_mat, w)))
```

3. Define the function calculating the efficient frontier:

```
def get_efficient_frontier(avg_rtns, cov_mat, rtns_range):  
  
    efficient_portfolios = []  
  
    n_assets = len(avg_returns)  
    args = (avg_returns, cov_mat)  
    bounds = tuple((0,1) for asset in range(n_assets))  
    initial_guess = n_assets * [1. / n_assets, ]  
  
    for ret in rtns_range:  
        constr = (  
            {"type": "eq",  
             "fun": lambda x: get_portf_rtn(x, avg_rtns) - ret},  
            {"type": "eq",  
             "fun": lambda x: np.sum(x) - 1}  
        )  
        ef_portf = sco.minimize(get_portf_vol,  
                               initial_guess,  
                               args=args, method="SLSQP",  
                               constraints=constr,  
                               bounds=bounds)  
        efficient_portfolios.append(ef_portf)  
  
    return efficient_portfolios
```

4. Define the considered range of expected portfolio returns:

```
rtns_range = np.linspace(-0.1, 0.55, 200)
```

5. Calculate the efficient frontier:

```
efficient_portfolios = get_efficient_frontier(avg_returns,  
                                              cov_mat,  
                                              rtns_range)
```

6. Extract the volatilities of the efficient portfolios:

```
vols_range = [x["fun"] for x in efficient_portfolios]
```

7. Plot the calculated efficient frontier, together with the simulated portfolios:

```

fig, ax = plt.subplots()
portf_results_df.plot(kind="scatter", x="volatility",
                      y="returns", c="sharpe_ratio",
                      cmap="RdYlGn", edgecolors="black",
                      ax=ax)
ax.plot(vols_range, rtns_range, "b--", linewidth=3)
ax.set(xlabel="Volatility",
       ylabel="Expected Returns",
       title="Efficient Frontier")

plt.show()

```

The following figure presents a graph of the efficient frontier, calculated using numerical optimization:

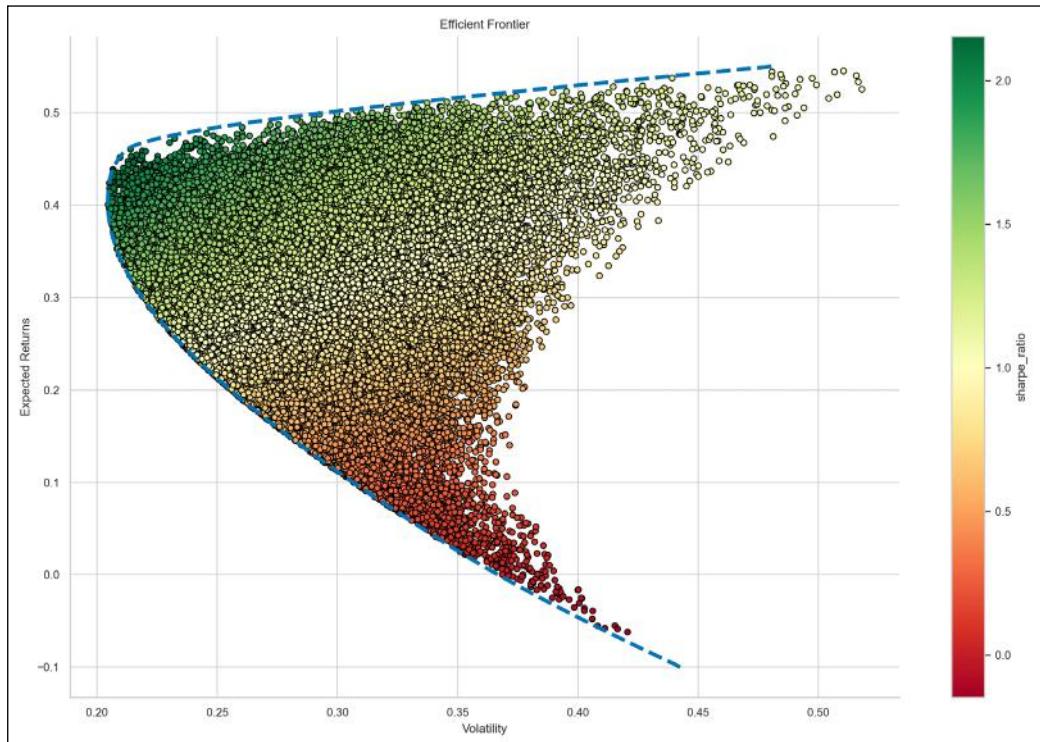


Figure 11.12: Efficient frontier identified using numerical optimization together with the previously generated random portfolios

We see that the efficient frontier has a very similar shape to the one obtained using Monte Carlo simulations. The only difference is that the line is smoother.

8. Identify the minimum volatility portfolio:

```
min_vol_ind = np.argmin(vols_range)
min_vol_portf_rtn = rtns_range[min_vol_ind]
min_vol_portf_vol = efficient_portfolios[min_vol_ind]["fun"]

min_vol_portf = {
    "Return": min_vol_portf_rtn,
    "Volatility": min_vol_portf_vol,
    "Sharpe Ratio": (min_vol_portf_rtn / min_vol_portf_vol)
}
```

9. Print the performance summary:

```
print_portfolio_summary(min_vol_portf,
                      efficient_portfolios[min_vol_ind]["x"],
                      ASSETS,
                      name="Minimum Volatility")
```

Running the snippet results in the following summary:

```
Minimum Volatility portfolio ----
Performance
Return: 40.30% Volatility: 20.45% Sharpe Ratio: 197.10%
Weights
META: 15.98% MSFT: 79.82% TSLA: 0.00% TWTR: 4.20%
```

The minimum volatility portfolio is achieved by investing mostly in Microsoft and Meta, while not investing in Tesla at all.

How it works...

As mentioned in the introduction, we continued the example from the previous recipe. That is why we had to run *Steps 1 to 4* from there (not shown here for brevity), to have all the required data. As an extra prerequisite, we had to import the optimization module from SciPy.

In *Step 2*, we defined two functions, which return the expected portfolio return and volatility, given historical data and the portfolio weights. We had to define these functions instead of calculating these metrics directly as we use them later on in the optimization procedure. The algorithm iteratively tries different weights and needs to be able to use the current values of the target variables (weights) to arrive at the metric it tries to optimize.

In *Step 3*, we defined a function called `get_efficient_frontier`. Its goal is to return a list containing the efficient portfolios, given historical metrics and the considered range of expected portfolio returns. This was the most important step of the recipe and contained a lot of nuances. We describe the logic of the function sequentially:

1. The outline of the function is that it runs the optimization procedure for each expected portfolio return in the considered range, and stores the resulting optimal portfolio in a list.
2. Outside of the `for` loop, we defined a couple of objects that we pass into the optimizer:
 - The arguments that are passed to the objective function. In this case, these are the historical average returns and the covariance matrix. The function that we optimize must accept the arguments as inputs. That is why we pass the returns to the `get_portf_vol` function (defined in *Step 2*), even though they are not necessary for calculations and are not used within the function.
 - `bounds` (a nested tuple)—for each target variable (weight), we provide a tuple containing the boundary values, that is, the minimum and maximum allowable values. In this case, the values span the range from 0 to 1 (no negative weights, as per the MPT).
 - `initial_guess`, which is the initial guess of the target variables. The goal of using the initial guess is to make the optimization run faster and more efficiently. In this case, the guess is the equally-weighted allocation.
3. Inside the `for` loop, we defined the last element used for the optimization—the constraints. We defined two constraints:
 - The expected portfolio return must be equal to the provided value.
 - The sum of the weights must be equal to 1.

The first constraint is the reason why the constraint's tuple is defined within the loop. That is because the loop passes over the considered range of expected portfolio returns, and for each value, we find the optimal risk level.

4. We run the optimizer with the **Sequential Least-Squares Programming (SLSQP)** algorithm, which is frequently used for generic minimization problems. For the function to be minimized, we pass the previously defined `get_portfolio_vol` function.



The optimizer sets the equality (`eq`) constraint to 0. That is why the intended constraint, `np.sum(weights) == 1`, is expressed as `np.sum(weights) - 1 == 0`.

In *Steps 4* and *5*, we defined the range of expected portfolio returns (based on the range we empirically observed in the previous recipe) and ran the optimization function.

In *Step 6*, we iterated over the list of efficient portfolios and extracted the optimal volatilities. We extracted the volatility from the `scipy.optimize.OptimizeResult` object by accessing the `fun` element. This stands for the optimized objective function which is, in this case, the portfolio volatility.

In Step 7, we added the calculated efficient frontier on top of the plot from the previous recipe, *Finding the efficient frontier using Monte Carlo simulations*. All the simulated portfolios lie on or below the efficient frontier, which is what we expected to happen.

In Steps 8 and 9, we identified the minimum volatility portfolio, printed the performance metrics, and showed the portfolio's weights (extracted from the efficient frontier).

We can now compare the two minimum volatility portfolios: the one obtained using Monte Carlo simulations, and the one we received from optimization. The prevailing pattern in the allocation is the same—allocate the majority of the available resources to Meta and Microsoft. We can also see that the volatility of the optimized strategy is slightly lower. This means that among the 100,000 portfolios, we have not simulated the actual minimum volatility portfolio (for the considered range of expected portfolio returns).

There's more...

We can also use the optimization approach to find the weights that generate a portfolio with the highest expected Sharpe ratio, that is, the tangency portfolio. To do so, we first need to redefine the objective function, which now will be the negative of the Sharpe ratio. The reason why we use the negative is that optimization algorithms run minimization problems. We can easily approach maximization problems by changing the sign of the objective function:

1. Define the new objective function (negative Sharpe ratio):

```
def neg_sharpe_ratio(w, avg_rtns, cov_mat, rf_rate):  
    portf_returns = np.sum(avg_rtns * w)  
    portf_volatility = np.sqrt(np.dot(w.T, np.dot(cov_mat, w)))  
    portf_sharpe_ratio = (  
        (portf_returns - rf_rate) / portf_volatility  
    )  
    return -portf_sharpe_ratio
```

The second step is very similar to what we have already done with the efficient frontier, this time without the `for` loop, as we are only searching for one set of weights. We include the risk-free rate in the arguments (though we assume it is 0%, for simplicity) and only use one constraint—the sum of the target variables must be equal to 1.

2. Find the optimized portfolio:

```
n_assets = len(avg_returns)  
RF_RATE = 0  
  
args = (avg_returns, cov_mat, RF_RATE)  
constraints = ({"type": "eq",  
               "fun": lambda x: np.sum(x) - 1})  
bounds = tuple((0,1) for asset in range(n_assets))  
initial_guess = n_assets * [1. / n_assets]
```

```
max_sharpe_portf = sco.minimize(neg_sharpe_ratio,
                                 x0=initial_guess,
                                 args=args,
                                 method="SLSQP",
                                 bounds=bounds,
                                 constraints=constraints)
```

3. Extract information about the maximum Sharpe ratio portfolio:

```
max_sharpe_portf_w = max_sharpe_portf["x"]
max_sharpe_portf = {
    "Return": get_portf_rtn(max_sharpe_portf_w, avg_returns),
    "Volatility": get_portf_vol(max_sharpe_portf_w,
                                 avg_returns,
                                 cov_mat),
    "Sharpe Ratio": -max_sharpe_portf["fun"]
}
```

4. Print the performance summary:

```
print_portfolio_summary(max_sharpe_portf,
                       max_sharpe_portf_w,
                       ASSETS,
                       name="Maximum Sharpe Ratio")
```

Running the snippet prints the following summary of the portfolio maximizing the Sharpe ratio:

```
Maximum Sharpe Ratio portfolio ----
Performance
Return: 45.90% Volatility: 21.17% Sharpe Ratio: 216.80%
Weights
META: 0.00% MSFT: 96.27% TSLA: 3.73% TWTR: 0.00%
```

To achieve the maximum Sharpe ratio, the investor should invest mostly in Microsoft (>96% allocation), with a 0% allocation to Meta and Twitter.

See also

- Markowitz, H., 1952. “Portfolio Selection,” *The Journal of Finance*, 7(1): 77–91

Finding the efficient frontier using convex optimization with CVXPY

In the previous recipe, *Finding the efficient frontier using optimization with SciPy*, we found the efficient frontier using numerical optimization with the SciPy library. We used portfolio volatility as the metric we wanted to minimize. However, it is also possible to state the same problem a bit differently and use convex optimization to find the efficient frontier.

We can reframe the mean-variance optimization problem into a risk-aversion framework, in which the investor wants to maximize the risk-adjusted return:

$$\begin{aligned} \max \quad & \omega^T \mu - \gamma \omega^T \Sigma \omega \\ \text{s.t.} \quad & \omega^T \mathbf{1} = 1 \\ & \omega \geq 0 \end{aligned}$$

Here, $\gamma \in [0, \infty)$ is the risk-aversion parameter, and the constraints specify that the weights must sum up to 1, and short-selling is not allowed. The higher the value of γ , the more risk-averse the investor is.



Short-selling assumes borrowing an asset and selling it on the open market. Then, we purchase the asset later at a lower price. Our gain is the difference after repaying the initial loan. In this recipe, we use the same data as in the previous two recipes, to make sure the results are comparable.

Getting ready

This recipe requires running all the code from the previous recipes:

- *Finding the efficient frontier using Monte Carlo simulations*
- *Finding the efficient frontier using optimization with SciPy*

How to do it...

Execute the following steps to find the efficient frontier using convex optimization:

1. Import the library:

```
import cvxpy as cp
```

2. Convert the annualized average returns and the covariance matrix to numpy arrays:

```
avg_returns = avg_returns.values  
cov_mat = cov_mat.values
```

3. Set up the optimization problem:

```

weights = cp.Variable(n_assets)
gamma_par = cp.Parameter(nonneg=True)
portf_rtn_cvx = avg_returns @ weights
portf_vol_cvx = cp.quad_form(weights, cov_mat)
objective_function = cp.Maximize(
    portf_rtn_cvx - gamma_par.*.portf_vol_cvx
)
problem = cp.Problem(
    objective_function,
    [cp.sum(weights) == 1, weights >= 0]
)

```

4. Calculate the efficient frontier:

```

N_POINTS = 25
portf_rtn_cvx_ef = []
portf_vol_cvx_ef = []
weights_ef = []
gamma_range = np.logspace(-3, 3, num=N_POINTS)

for gamma in gamma_range:
    gamma_par.value = gamma
    problem.solve()
    portf_vol_cvx_ef.append(cp.sqrt(portf_vol_cvx).value)
    portf_rtn_cvx_ef.append(portf_rtn_cvx.value)
    weights_ef.append(weights.value)

```

5. Plot the allocation for different values of the risk-aversion parameter:

```

weights_df = pd.DataFrame(weights_ef,
                           columns=ASSETS,
                           index=np.round(gamma_range, 3))
ax = weights_df.plot(kind="bar", stacked=True)
ax.set(title="Weights allocation per risk-aversion level",
       xlabel=r"\$\\gamma\$",
       ylabel="weight")
ax.legend(bbox_to_anchor=(1,1))

```

In *Figure 11.13*, we can see the asset allocation for the considered range of risk-aversion parameters (γ):

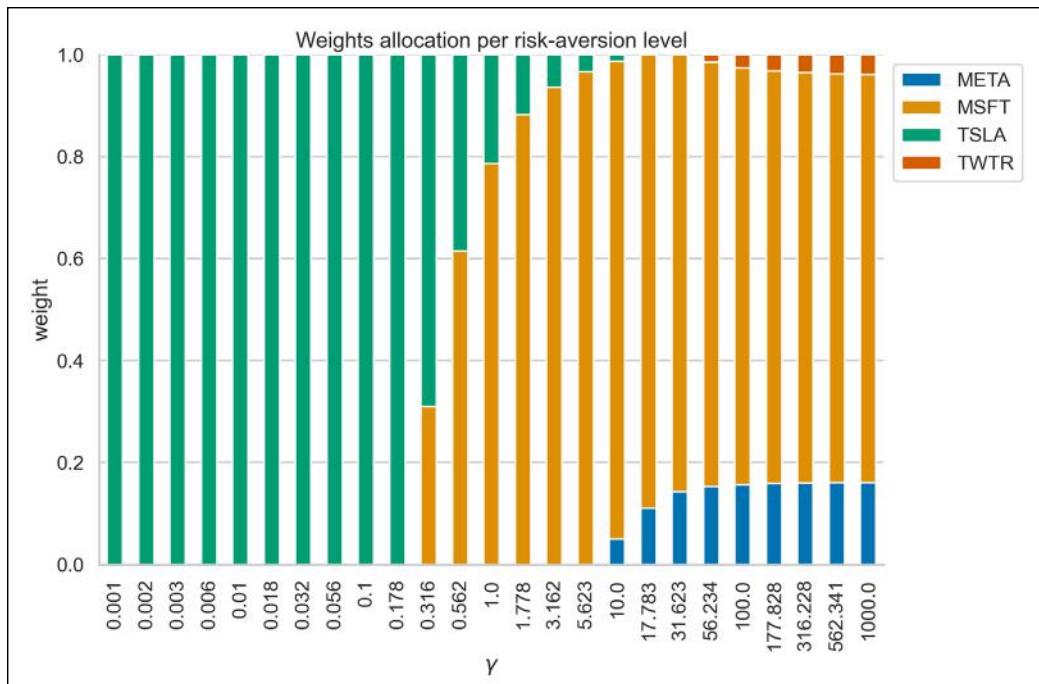


Figure 11.13: Asset allocation per various levels of risk-aversion

In Figure 11.13, we can see that for very small values of γ , the investor would allocate 100% of their resources to Tesla. As we increased the risk aversion, the allocation to Tesla grew smaller, and more weight was allocated to Microsoft and the other assets. At the other end of the considered values for the parameter, the investor would allocate 0% to Tesla.

- Plot the efficient frontier, together with the individual assets:

```
fig, ax = plt.subplots()
ax.plot(portf_vol_cvx_ef, portf_rtn_cvx_ef, "g-")
for asset_index in range(n_assets):
    plt.scatter(x=np.sqrt(cov_mat[asset_index, asset_index]),
                y=avg_returns[asset_index],
                marker=MARKERS[asset_index],
                label=ASSETS[asset_index],
                s=150)
ax.set(title="Efficient Frontier",
       xlabel="Volatility",
       ylabel="Expected Returns")
ax.legend()
```

Figure 11.14 presents the efficient frontier, generated by solving the convex optimization problem.

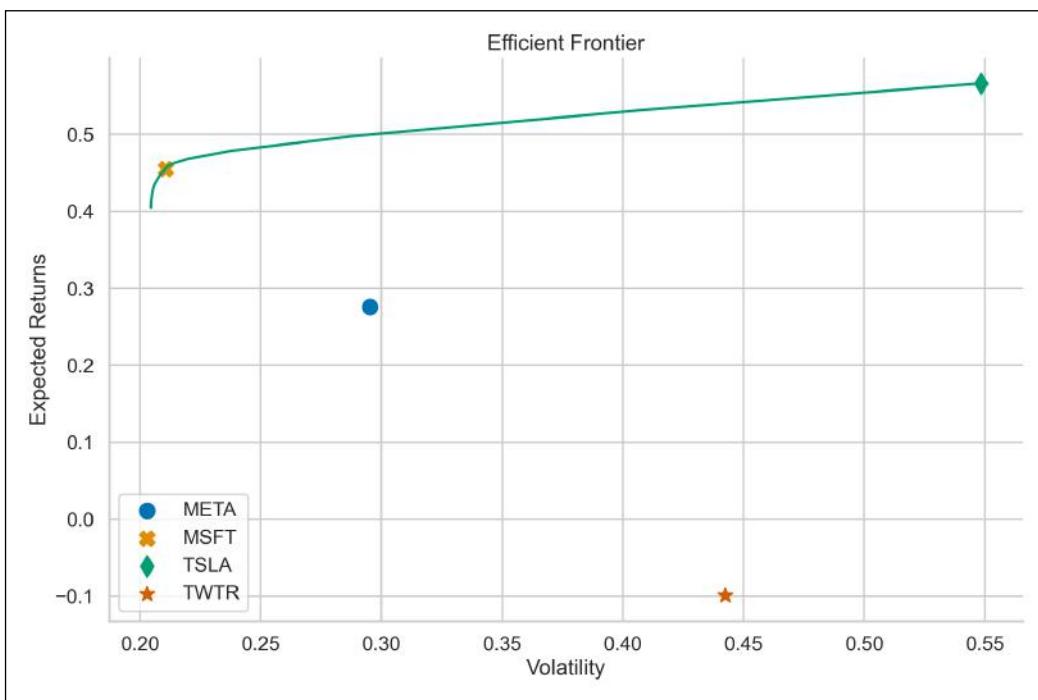


Figure 11.14: Efficient frontier identified by solving the convex optimization problem

The generated frontier is similar to the one in Figure 11.10 (generated using Monte Carlo simulations). Back then, we established that a portfolio consisting of only Microsoft's stocks lies very close to the efficient frontier. Now we can say the same about the portfolio comprised entirely of Tesla's stocks. When using Monte Carlo simulations, we did not have enough observations generated in that part of the returns/volatility plane to draw the efficient frontier line around that portfolio. In the *There's more...* section, we also compare this frontier to the one obtained in the previous recipe, in which we used the SciPy library.

How it works...

As mentioned in the introduction, we continued the example from the previous two recipes. That is why we had to run *Steps 1 to 4* from the *Finding the efficient frontier using Monte Carlo simulations* recipe (not shown here for brevity) to have all the required data. As an extra step, we had to import the cvxpy convex optimization library. We additionally converted the historical average returns and the covariance matrix into numpy arrays.

In *Step 3*, we set up the optimization problem. We started by defining the target variables (`weights`), the risk-aversion parameter (`gamma_par`, where “par” is added to highlight that it is a parameter of the optimization routine), the portfolio returns and volatility (both using the previously defined `weights` object), and lastly, the objective function—the risk-adjusted returns we want to maximize. Then, we created the `cp.Problem` object and passed the objective function and a list of constraints as arguments.



We used `cp.quad_form(x, y)` to express the following multiplication: $x^T y x$.

In *Step 4*, we found the efficient frontier by solving the convex optimization problem for multiple values of the risk-aversion parameter. To define the considered values, we used the `np.logspace` function to get 25 values of γ . For each value of the parameter, we found the optimal solution by running `problem.solve()`. We stored the values of interest in dedicated lists.



`np.logspace` is similar to `np.linspace`; the difference is that the former finds numbers evenly spread on a log scale instead of a linear scale.

In *Step 5*, we plotted the asset allocation per various levels of risk aversion. Lastly, we plotted the efficient frontier, together with the individual assets.

There's more...

Comparing the results from two formulations of the asset allocation problem

We can also plot the two efficient frontiers for comparison—the one calculated by minimizing the volatility per expected level of return, and the other one using convex optimization and maximizing the risk-adjusted return:

```
x_lim = [0.2, 0.6]
y_lim = [0.4, 0.6]

fig, ax = plt.subplots(1, 2)
ax[0].plot(vols_range, rtns_range, "g-", linewidth=3)
ax[0].set(title="Efficient Frontier - Minimized Volatility",
           xlabel="Volatility",
           ylabel="Expected Returns",
           xlim=x_lim,
           ylim=y_lim)

ax[1].plot(portf_vol_cvx_ef, portf_rtn_cvx_ef, "g-", linewidth=3)
ax[1].set(title="Efficient Frontier - Maximized Risk-Adjusted Return",
           xlabel="Volatility",
           ylabel="Expected Returns",
           xlim=x_lim,
           ylim=y_lim)
```

Executing the snippet generates the following plots:

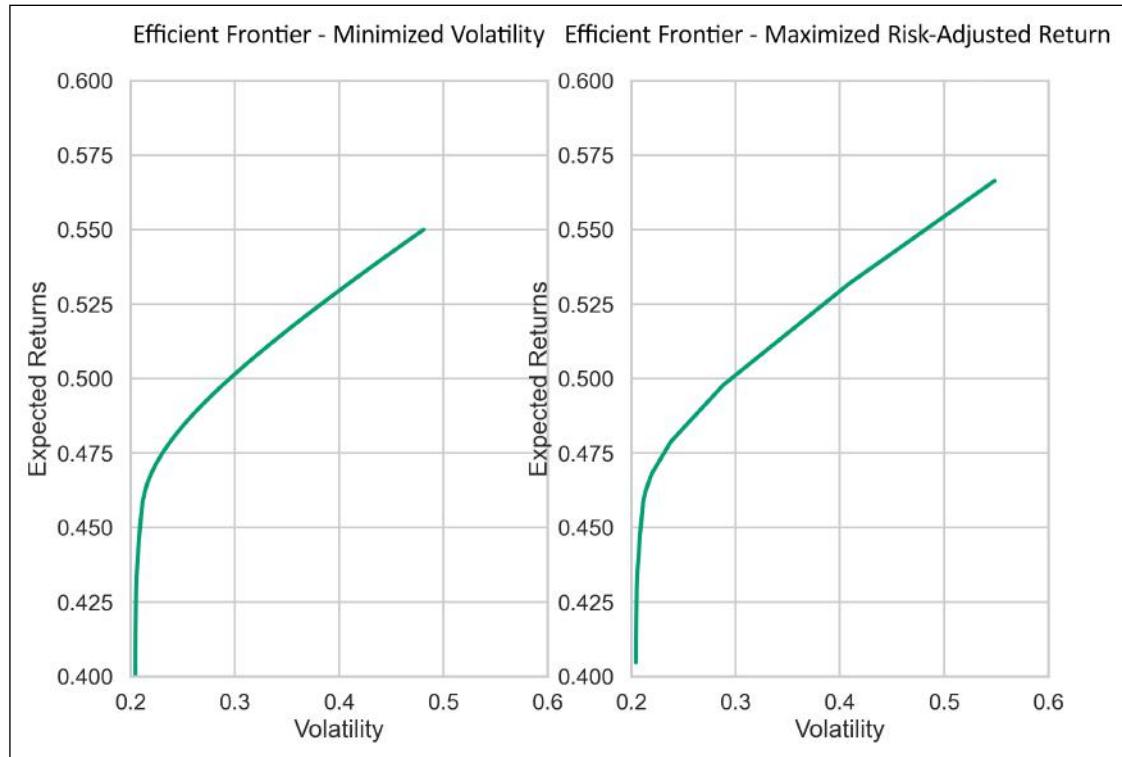


Figure 11.15: Comparison of efficient frontiers generated by minimizing volatility per expected level of return (left) and by maximizing the risk-adjusted return (right)

As we can see, the generated efficient frontiers are very similar, with some minor differences. First, the one obtained using minimization is smoother, as we used more points to calculate the frontier. Second, the right one is defined for a slightly larger range of possible volatility/returns pairs.

Allowing for leverage

Another interesting concept we can incorporate into the analysis is the maximum allowable leverage. We replace the non-negativity constraints on the weights with a max leverage constraint, using the norm of a vector.

In the following snippet, we only show what was added on top of the things we defined in *Step 3*:

```
max_leverage = cp.Parameter()
prob_with_leverage = cp.Problem(objective_function,
                                  [cp.sum(weights) == 1,
                                   cp.norm(weights, 1) <= max_leverage])
```

In the next snippet, we modify the code, this time to include two loops—one over potential values of the risk-aversion parameter, and the other one indicating the maximum allowable leverage. Max leverage equal to 1 (meaning no leverage) results in a case similar to the previous optimization problem (only this time, there is no non-negativity constraint).

We also redefine the placeholder objects (used for storing the results) to be either 2D matrices (`np.ndarrays`) or including the third dimension, in the case of weights.

```
LEVERAGE_RANGE = [1, 2, 5]
len_leverage = len(LEVERAGE_RANGE)
N_POINTS = 25

portf_vol_l = np.zeros((N_POINTS, len_leverage))
portf_rtn_l = np.zeros((N_POINTS, len_leverage))
weights_ef = np.zeros((len_leverage, N_POINTS, n_assets))

for lev_ind, leverage in enumerate(LEVERAGE_RANGE):
    for gamma_ind in range(N_POINTS):
        max_leverage.value = leverage
        gamma_par.value = gamma_range[gamma_ind]
        prob_with_leverage.solve()
        portf_vol_l[gamma_ind, lev_ind] = cp.sqrt(portf_vol_cvx).value
        portf_rtn_l[gamma_ind, lev_ind] = portf_rtn_cvx.value
        weights_ef[lev_ind, gamma_ind, :] = weights.value
```

In the following snippet, we plot the efficient frontiers for different maximum leverages. We can clearly see that higher leverage increases returns and, at the same time, allows for greater volatility.

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

for leverage_index, leverage in enumerate(LEVERAGE_RANGE):
    plt.plot(portf_vol_l[:, leverage_index],
              portf_rtn_l[:, leverage_index],
              label=f"{leverage}")

ax.set(title="Efficient Frontier for different max leverage",
       xlabel="Volatility",
       ylabel="Expected Returns")
ax.legend(title="Max leverage")
```

Executing the code generates the following figure.

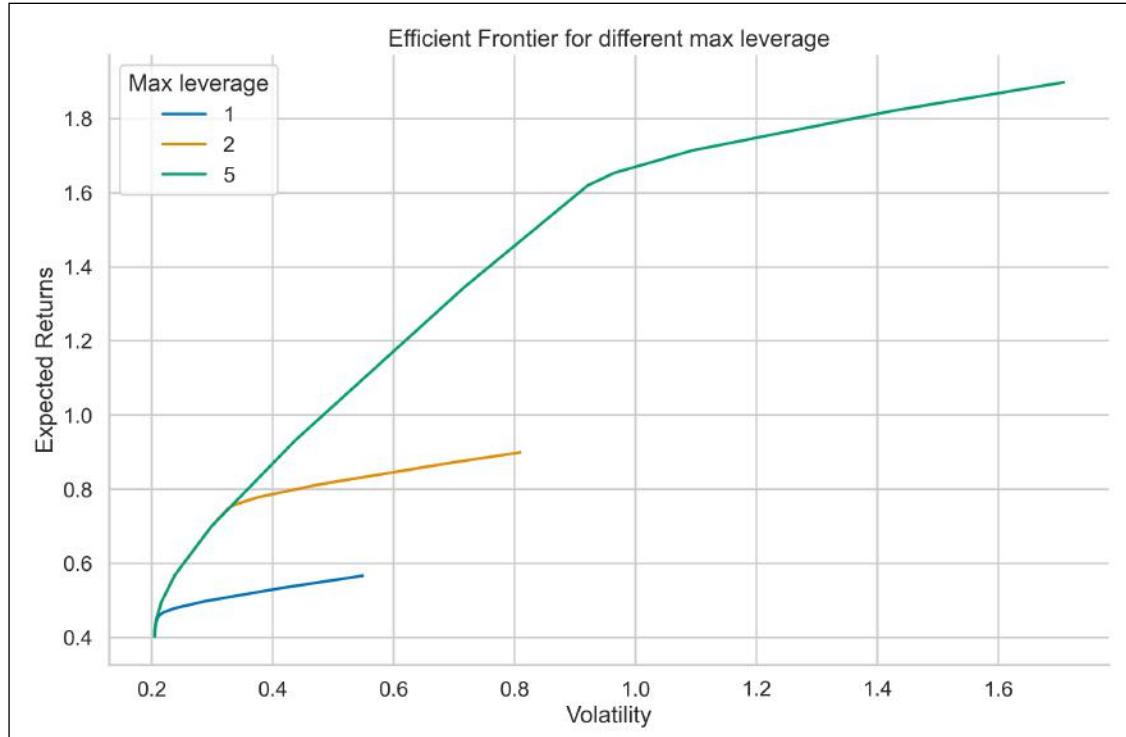


Figure 11.16: Efficient frontier for different values of maximum leverage

Lastly, we also recreate the plot showing weight allocation per varying risk-aversion levels. With a maximum leverage of 1, there is no short selling.

```
fig, ax = plt.subplots(len_leverage, 1, sharex=True)

for ax_index in range(len_leverage):
    weights_df = pd.DataFrame(weights_ef[ax_index],
                               columns=ASSETS,
                               index=np.round(gamma_range, 3))
    weights_df.plot(kind="bar",
                    stacked=True,
                    ax=ax[ax_index],
                    legend=None)
    ax[ax_index].set(
        ylabel=(f"max_leverage = {LEVERAGE_RANGE[ax_index]}"
               "\n weight")
    )
```

```

ax[len_leverage - 1].set(xlabel=r"\gamma")
ax[0].legend(bbox_to_anchor=(1,1))
ax[0].set_title("Weights allocation per risk aversion level",
                 fontsize=16)

```

Executing the snippet generates the following figure.

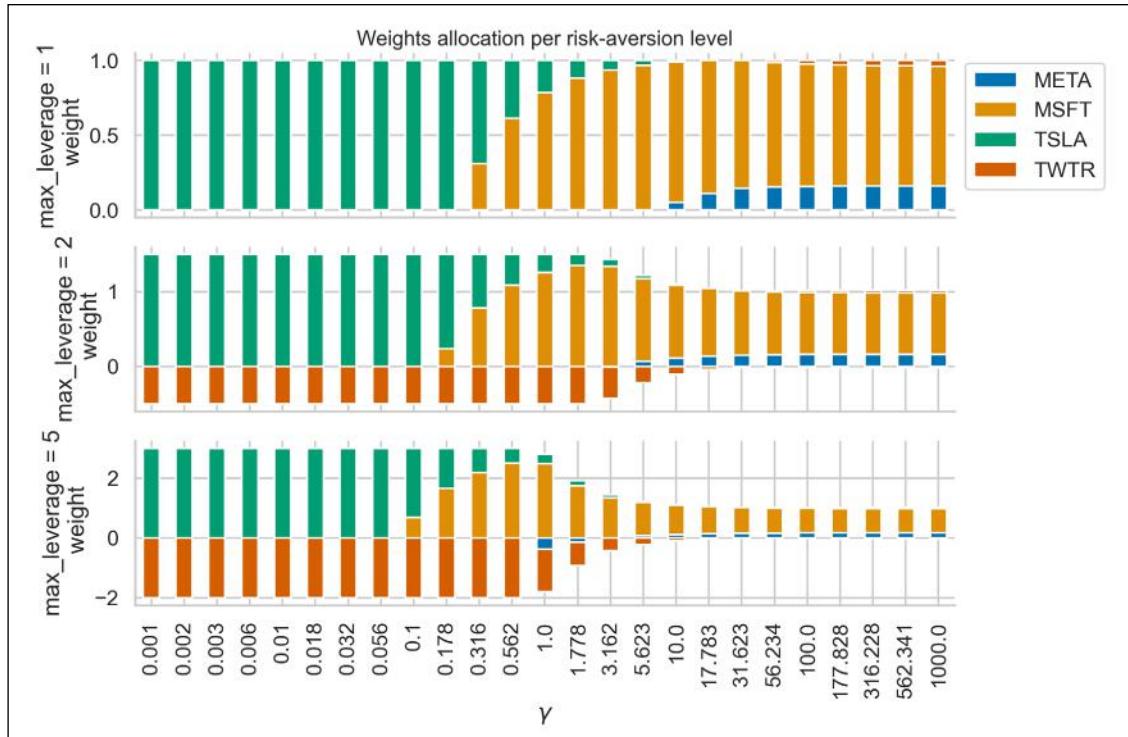


Figure 11.17: Asset allocation per different levels of risk aversion and maximum leverage

We can spot a clear pattern: with an increase in risk aversion, investors stop using leverage altogether and converge to a similar allocation for all levels of the maximum permitted leverage.

Finding the optimal portfolio with Hierarchical Risk Parity

De Prado (2018) explains that quadratic optimizers tend to deliver unreliable solutions, due to their instability, concentration, and underperformance. The main reason for all those troubles is the need to invert the covariance matrix, which is prone to cause large errors when the matrix is numerically ill-conditioned. He also refers to **Markowitz's curse**, which implies that the more correlated the investments are, the greater the need for diversification, which in turn leads to bigger estimation errors in the portfolio weights.

A potential solution is to introduce a hierarchical structure, which means that small estimation errors will no longer lead to entirely different allocations. That is possible because the quadratic optimizers have complete freedom to fully reshuffle the weights to their liking (unless some explicit constraints are enforced).

Hierarchical Risk Parity (HRP) is a novel portfolio optimization method that combines graph theory and machine learning techniques in order to build a diversified portfolio based on the information available in the covariance matrix. At a very high level, the algorithm works as follows:

1. Calculate a distance matrix based on the correlation of the assets (covariance matrix).
2. Cluster the assets into a tree structure with hierarchical clustering (based on the distance matrix).
3. Calculate the minimum variance portfolio within each branch of the tree.
4. Iterate over the levels of the tree and combine the portfolios at each node.

For a more detailed description of the algorithm, please refer to De Prado (2018).

We also mention some of the advantages of the HRP approach:

- It fully utilizes the information from the covariance matrix and does not require inverting it.
- It treats clustered assets as complements, rather than substitutes.
- The weights produced by the algorithm are more stable and robust.
- The solution can be intuitively understood with the help of visualizations.
- We can include additional constraints.
- Literature suggests that the method outperforms the classical mean-variance approaches out-of-sample.

In this recipe, we apply the Hierarchical Risk Parity algorithm to form a portfolio from the stocks of the 10 biggest US tech companies.

How to do it...

Execute the following steps to find the optimal asset allocation using the HRP:

1. Import the libraries:

```
import yfinance as yf
import pandas as pd
from pypfopt.expected_returns import returns_from_prices
from pypfopt.hierarchical_portfolio import HRPOpt
from pypfopt.discrete_allocation import (DiscreteAllocation,
                                         get_latest_prices)
from pypfopt import plotting
```

2. Download the stock prices of the 10 biggest US tech companies:

```
ASSETS = ["AAPL", "MSFT", "AMZN", "GOOG", "META",
          "V", "NVDA", "MA", "PYPL", "NFLX"]

prices_df = yf.download(ASSETS,
                       start="2021-01-01",
                       end="2021-12-31",
                       adjusted=True)
prices_df = prices_df["Adj Close"]
```

3. Calculate the returns from prices:

```
rtn_df = returns_from_prices(prices_df)
```

4. Find the optimal allocation using Hierarchical Risk Parity:

```
hrp = HRPOpt(returns=rtn_df)
hrp.optimize()
```

5. Display the (cleaned) weights:

```
weights = hrp.clean_weights()
print(weights)
```

This returns the following portfolio weights:

```
OrderedDict([('AAPL', 0.12992), ('AMZN', 0.156), ('META', 0.08134),
             ('GOOG', 0.08532), ('MA', 0.10028), ('MSFT', 0.1083), ('NFLX', 0.10164),
             ('NVDA', 0.04466), ('PYPL', 0.05326), ('V', 0.13928)])
```

6. Calculate the portfolio performance:

```
hrp.portfolio_performance(verbose=True, risk_free_rate=0);
```

which returns the following evaluation metrics:

```
Expected annual return: 23.3%
Annual volatility: 19.2%
Sharpe Ratio: 1.21
```

7. Visualize the hierarchical clustering used for finding the portfolio weights:

```
fig, ax = plt.subplots()
plotting.plot_dendrogram(hrp, ax=ax)
ax.set_title("Dendrogram of cluster formation")
plt.show()
```

Running the snippet generates the following plot:

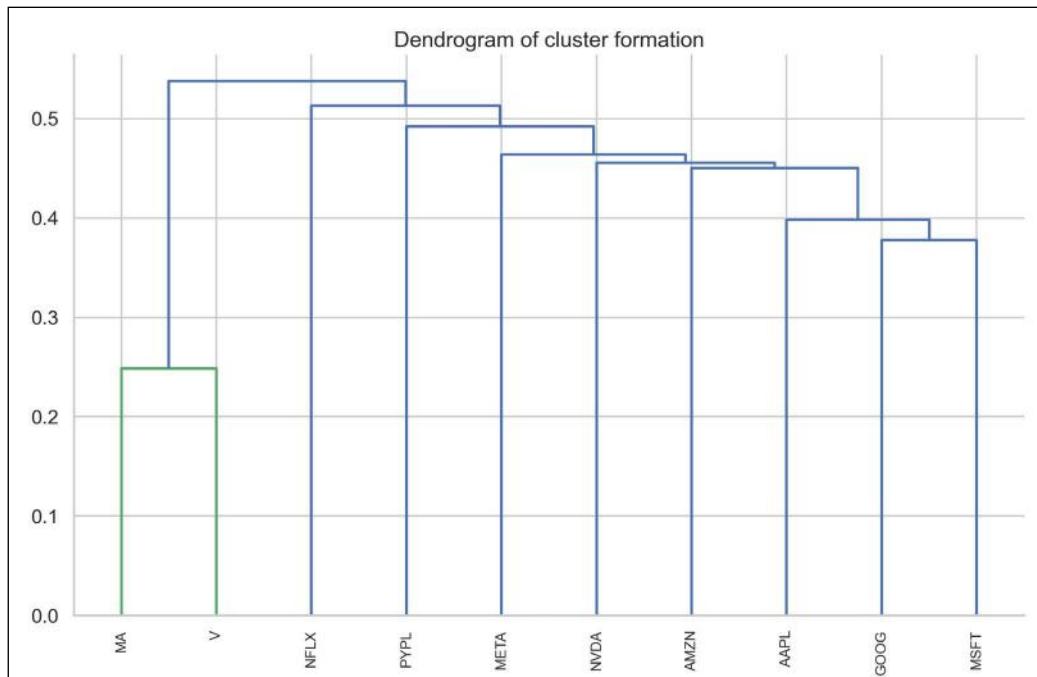


Figure 11.18: Dendrogram visualizing the process of cluster formation

In *Figure 11.18*, we can see that companies such as Visa and MasterCard were clustered together. In the plot, the y-axis represents the distance between the two leaves that are to be merged.

This makes sense, as if we wanted to invest in a publicly traded US credit card company like Visa, we might consider adding or reducing the allocation to another very similar company, such as MasterCard. Similarly in the case of Google and Microsoft, although the difference between those two companies is larger. This is the very idea of applying the hierarchy structure to the correlation between the assets.

8. Find the number of stocks to buy using 50,000 USD:

```
latest_prices = get_latest_prices(prices_df)
allocation_finder = DiscreteAllocation(weights,
                                         latest_prices,
                                         total_portfolio_value=50000)

allocation, leftover = allocation_finder.lp_portfolio()
print(allocation)
print(leftover)
```

Running the snippet prints the following dictionary of the suggested number of stocks to purchase and the leftover cash:

```
{'AAPL': 36, 'AMZN': 2, 'META': 12, 'GOOG': 2, 'MA': 14, 'MSFT': 16, 'NFLX': 8,  
'NVDA': 7, 'PYPL': 14, 'V': 31}  
12.54937744140625
```

How it works...

After importing the libraries, we downloaded the stock prices of the 10 largest US tech companies for the year 2021. In *Step 3*, we created a DataFrame containing the daily stock returns using the `returns_from_prices` function.

In *Step 4*, we instantiated the `HRP0pt` object and passed in the stock returns as input. Then, we used the `optimize` method to find the optimal weights. An inquisitive reader might notice that when describing the algorithm, we mentioned that it is based on the covariance matrix, while we used the return series as input. Under the hood, when we pass in the `returns` argument, the class computes the covariance matrix for us. Alternatively, we can pass in the covariance matrix directly using the `cov_matrix` argument.



When passing the covariance matrix directly, we can benefit from using alternative formulations of the covariance matrix, rather than the sample covariance. For example, we could use the Ledoit-Wolf shrinkage or the **oracle approximating shrinkage (OAS)**. You can find references for those methods in the *See also* section.

Then, we displayed the cleaned weights using the `clean_weights` method. It is a helper method that rounds the weights to 5 decimals (can be adjusted) and cuts off any weights below a certain threshold to 0. In *Step 6*, we calculated the portfolio's expected performance using the `portfolio_performance` method. While doing so, we changed the default risk-free rate to 0%.

In *Step 7*, we plotted the results of the hierarchical clustering using the `plot_dendrogram` function. The figure produced by this function is very useful for getting an understanding of how the algorithm works and which assets were clustered together.

In *Step 8*, we performed a discrete allocation based on the calculated weights. We assumed we had 50,000 USD and wanted to allocate as much as possible using the HRP weights. First, we recovered the latest prices from the downloaded prices, so the ones from 2021-12-30. Then, we instantiated an object of the `DiscreteAllocation` class by providing the weights, latest prices, and our budget. Lastly, we used the `lp_portfolio` method to use linear programming to find the number of stocks we should buy, while keeping in mind our budget. We obtained two objects as the output: a dictionary containing the pairs of assets and the corresponding number of stocks, and the remaining money.



An alternative approach to linear programming would be to employ the greedy iterative search, available under the `greedy_portfolios` method.

There's more...

PyPortfolioOpt has much more to offer than we have covered. For example, it greatly simplifies obtaining the efficient frontier. We can calculate it using the following steps:

1. Import the libraries:

```
from pypfopt.expected_returns import mean_historical_return
from pypfopt.risk_models import CovarianceShrinkage
from pypfopt.efficient_frontier import EfficientFrontier
from pypfopt.plotting import plot_efficient_frontier
```

2. Get the expected returns and the covariance matrix:

```
mu = mean_historical_return(prices_df)
S = CovarianceShrinkage(prices_df).ledoit_wolf()
```

As we have already established multiple times in this chapter, mean-variance optimization requires two components: the expected returns of the assets and their covariance matrix. PyPortfolioOpt offers multiple possibilities for calculating both of them. While we have already mentioned alternatives to the covariance matrix, you can use the following for the expected returns: historical mean return, exponentially weighted mean historical return, and CAPM estimate of returns. Here, we calculated the historical mean and the Ledoit-Wolf shrinkage estimate of the covariance matrix.

3. Find and plot the efficient frontier:

```
ef = EfficientFrontier(mu, S)

fig, ax = plt.subplots()
plot_efficient_frontier(ef, ax=ax, show_assets=True)
ax.set_title("Efficient Frontier")
```

Running the snippet generates the following figure:

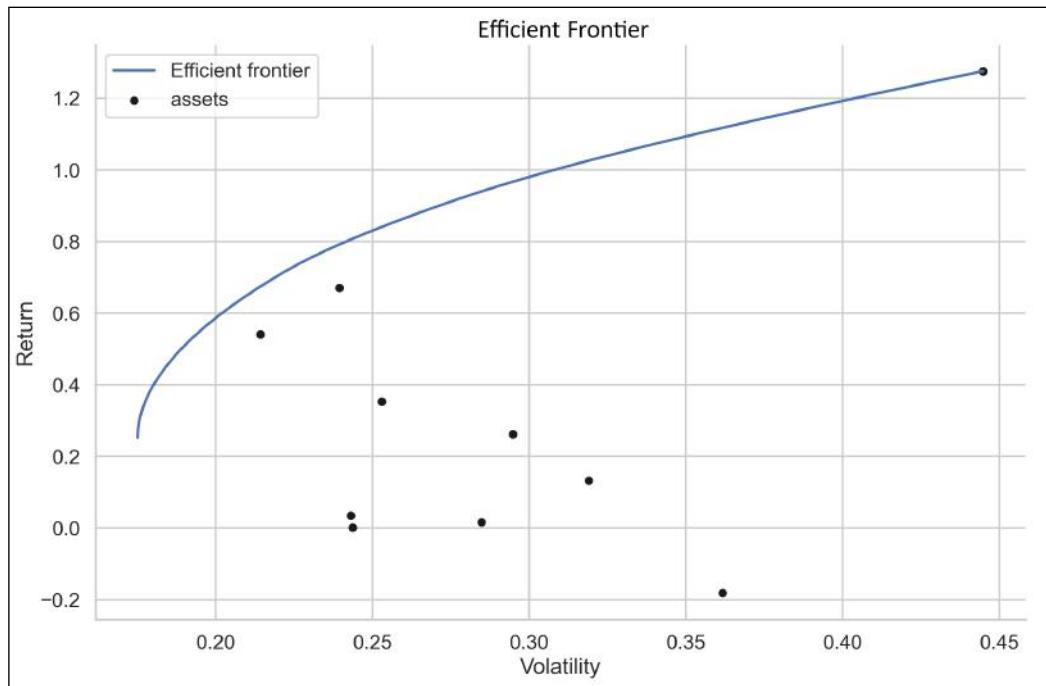


Figure 11.19: Efficient frontier obtained using the Ledoit-Wolf shrinkage estimate of the covariance matrix

4. Identify the tangency portfolio:

```
ef = EfficientFrontier(mu, S)
weights = ef.max_sharpe(risk_free_rate=0)
print(ef.clean_weights())
```

This returns the following portfolio weights:

```
OrderedDict([('AAPL', 0.0), ('AMZN', 0.0), ('META', 0.0), ('GOOG', 0.55146), ('MA', 0.0), ('MSFT', 0.11808), ('NFLX', 0.0), ('NVDA', 0.33046), ('PYPL', 0.0), ('V', 0.0)])
```

The `EfficientFrontier` class allows for identifying more than just the tangency portfolio. We can also use the following methods:

- `min_volatility`: Finds the portfolio with minimum volatility.
- `max_quadratic_utility`: Finds the portfolio that maximizes the quadratic utility, given a level of risk aversion. This is the same approach as the one we have covered in the previous recipe.
- `efficient_risk`: Finds a portfolio that maximizes the return for a given target risk.
- `efficient_return`: Finds a portfolio that minimizes the risk for a given target return.

For the last two options, we can generate market neutral portfolios, that is, portfolios with weights summing up to zero.

As we have mentioned before, the functionalities we showed are just the proverbial tip of the iceberg. Using the library, we can also explore the following:

- Incorporate sector constraints: Let's assume you want to have a portfolio of stocks from various sectors, while keeping some conditions, for example, having at least 20% in tech.
- Optimize for transaction costs: In a case when we already have a portfolio and want to rebalance, it might be quite expensive to completely rebalance the portfolio (and as we have discussed before, the instability of the portfolio weights can be a big disadvantage of the mean-variance optimization). In such a case, we can add an additional objective to rebalance the portfolio while keeping the transaction costs as low as possible.
- Use the L2 regularization while optimizing the portfolio: By using the regularization we counter the behavior of many weights dropping to zero. We can experiment with different values of the gamma parameter to find the allocation that works best for us. You might already be familiar with the L2 regularization thanks to the famous Ridge Regression algorithm.
- Use the Black-Litterman model to get a more stable model of the expected returns than just by using the historical mean returns. It is a Bayesian approach to asset allocation, which combines a prior estimate of returns with views on certain assets to arrive at a posterior estimate of expected returns.

In the notebook on GitHub, you can also find short examples of finding the efficient frontier while allowing for short-selling or using L2 regularization.



You can also experiment with not using the expected returns. Literature suggests that due to the difficulties in getting an accurate estimate of expected returns, minimum variance portfolios consistently outperform the maximum Sharpe ratio portfolios out-of-sample.

See also

Additional resources concerning the approaches mentioned in the recipe:

- Black, F; & Litterman, R. 1991. "Combining investor views with market equilibrium," *The Journal of Fixed Income*, 1, (2): 7-18: <https://doi.org/10.3905/jfi.1991.408013>

- Black, F., & Litterman, R. 1992. “Global portfolio optimization,” *Financial Analysts Journal*, 48(5): 28-43
- Chen, Y., Wiesel, A., Eldar, Y. C., & Hero, A. O. 2010. “Shrinkage Algorithms for MMSE Covariance Estimation,” *IEEE Transactions on Signal Processing*, 58(10): 5016-5029: <https://doi.org/10.1109/TSP.2010.2053029>
- De Prado, M. L. 2016. “Building diversified portfolios that outperform out of sample,” *The Journal of Portfolio Management*, 42(4): 59-69: <https://doi.org/10.3905/jpm.2016.42.4.059>.
- De Prado, M. L. 2018. *Advances in Financial Machine Learning*. John Wiley & Sons
- Ledoit, O., & Wolf, M. 2003 “Improved estimation of the covariance matrix of stock returns with an application to portfolio selection,” *Journal of Empirical Finance*, 10(5): 603-621
- Ledoit, O., & Wolf, M. 2004. “Honey, I shrunk the sample covariance matrix,” *The Journal of Portfolio Management*, 30(4): 110-119: <https://doi.org/10.3905/jpm.2004.110>

Summary

In this chapter, we have learned about asset allocation. We started with the simplest equally-weighted portfolio, which was proven to be quite difficult to outperform, even with advanced optimization techniques. Then, we explored various approaches to calculating the efficient frontier using mean-variance optimization. Lastly, we also touched upon some of the recent developments in asset allocation, that is, the Hierarchical Risk Parity algorithm.

You might find the following references interesting for learning more about approaching asset allocation with Python:

- [Riskfolio-Lib](https://github.com/dcajasn/Riskfolio-Lib) (<https://github.com/dcajasn/Riskfolio-Lib>): Another popular portfolio optimization library containing a wide selection of algorithms and evaluation metrics.
- [deepdow](https://github.com/jankrepl/deepdow) (<https://github.com/jankrepl/deepdow>): A Python library connecting portfolio optimization and deep learning.

In the next chapter, we cover various methods of backtesting trading and asset allocation strategies.

12

Backtesting Trading Strategies

In the previous chapters, we gained the knowledge necessary to create trading strategies. On the one hand, we could use technical analysis to identify trading opportunities. On the other, we could use some of the other techniques we have already covered in the book. We could try to use knowledge about factor models or volatility forecasting. Or, we could use portfolio optimization techniques to determine the optimal quantity of assets for our investment. One crucial thing that is still missing is evaluating how such a strategy would have performed if we had implemented it in the past. That is the goal of backtesting, which we explore in this chapter.

Backtesting can be described as a realistic simulation of our trading strategy, which assesses its performance using historical data. The underlying idea is that the backtest performance should be indicative of future performance when the strategy is actually used on the market. Naturally, this will not always be the case and we should keep that in mind when experimenting.

There are multiple ways of approaching backtesting, however, we should always remember that a backtest should faithfully represent how markets operate, how trades are executed, what orders are available, and so on. For example, forgetting to account for transaction costs can quickly turn a “profitable” strategy into a failed experiment.

We have already mentioned the generic uncertainty around predictions in the ever-changing financial markets. However, there are also some implementation aspects that can bias the results of backtests and increase the risk of confusing in-sample performance with generalizable patterns that will also hold out of sample. We briefly mention some of those below:

- **Look-ahead bias:** This potential flaw emerges when we develop a trading strategy using historical data before it was actually known/available. Some examples include corrections of reported financial statements after their publication, stock splits, or reverse splits.
- **Survivorship bias:** This bias arises when we backtest only using data about securities that are currently active/tradeable. By doing so, we omit the assets that have disappeared over time (due to bankruptcy, delisting, acquisition, and so on). Most of the time, those assets did not perform well and our strategies can be skewed by failing to include those, as those assets could have been picked up in the past when they were still available in the markets.

- **Outlier detection and treatment:** The main challenge is to discern the outliers that are not representative of the analyzed period as opposed to the ones that are an integral part of the market behavior.
- **Representative sample period:** As the goal of the backtest is to provide an indication of future performance, the sample data should reflect the current, and potentially also future, market behavior. By not spending enough time on this part, we can miss some crucial market regime aspects such as volatility (too few/many extreme events) or volume (too few data points).
- **Meeting investment objectives and constraints over time:** It can happen that a strategy leads to good performance at the very end of the evaluation period. However, in some periods when it was active, it resulted in unacceptably high losses or volatility. We could potentially track those by using rolling performance/risk metrics, for example, the value-at-risk or the Sharpe/Sortino ratio.
- **Realistic trading environment:** We have already mentioned that failing to include transaction costs can greatly impact the end result of a backtest. What is more, real-life trading involves further complications. For example, it might not be possible to execute all trades at all times or at the target price. Some of the things to consider are **slippage** (the difference between the expected price of a trade and the price at which the trade is executed), the availability of a counterparty for short positions, broker fees, and so on. The realistic environment also accounts for the fact that we might make a trading decision based on the close prices of one day, but the trade will be (potentially) executed based on the open prices of the next trading day. It can happen that the order we prepare will not be executed due to large price differences.
- **Multiple testing:** When running multiple backtests, we might discover spurious results or a strategy that overfits the test sample and produces suspiciously positive results that are unlikely to hold for out-of-sample data encountered during live trading. Also, we might leak prior knowledge of what works and what does not into the design of strategies, which can lead to further overfitting. Some things that we can consider are: reporting the number of trials, calculating the minimum backtest length, using some sort of optimal stopping rule, or calculating metrics that account for the effect of multiple testing (for example, the deflated Sharpe ratio).

In this chapter, we show how to run backtests of various trading strategies using two approaches: vectorized and event-driven. We will go into the details of each of the approaches later on, but now we can state that the first one works well for a quick test to see if there is any potential in the strategy. On the other hand, the latter is more suited for thorough and rigorous testing, as it tries to account for many of the potential issues mentioned above.

The key learning of this chapter is how to set up a backtest using popular Python libraries. We will be showing a few examples of strategies built on the basis of popular technical indicators or a strategy using mean-variance portfolio optimization. With that knowledge, you can backtest any strategy you can come up with.

We present the following recipes in this chapter:

- Vectorized backtesting with pandas
- Event-driven backtesting with backtrader
- Backtesting a long/short strategy based on the RSI
- Backtesting a buy/sell strategy based on Bollinger bands
- Backtesting a moving average crossover strategy using crypto data
- Backtesting a mean-variance portfolio optimization

Vectorized backtesting with pandas

As we mentioned in the introduction to this chapter, there are two approaches to carrying out backtests. The simpler one is called **vectorized backtesting**. In this approach, we multiply a signal vector/matrix (containing an indicator of whether we are entering or closing a position) by the vector of returns. By doing so, we calculate the performance over a certain period of time.

Due to its simplicity, this approach cannot deal with many of the issues we described in the introduction, for example:

- We need to manually align the timestamps to avoid look-ahead bias.
- There is no explicit position sizing.
- All performance measurements are calculated manually at the very end of the backtest.
- Risk-management rules like stop-loss are not easy to incorporate.

That is why we should use vectorized backtesting mostly if we are dealing with simple trading strategies and want to explore their initial potential in a few lines of code.

In this recipe, we backtest a very simple strategy with the following set of rules:

- We enter a long position if the close price is above the 20-day Simple Moving Average (SMA)
- We close the position when the close price goes below the 20-day SMA
- Short selling is not allowed
- The strategy is unit agnostic (we can enter a position of 1 share or 1000 shares) because we only care about the percentage change in the prices

We backtest this strategy using Apple's stock and its historical prices from the years 2016 to 2021.

How to do it...

Execute the following steps to backtest a simple strategy using the vectorized approach:

1. Import the libraries:

```
import pandas as pd
import yfinance as yf
import numpy as np
```

2. Download Apple's stock prices from the years 2016 to 2021 and keep only the adjusted close price:

```
df = yf.download("AAPL",
                  start="2016-01-01",
                  end="2021-12-31",
                  progress=False)
df = df[["Adj Close"]]
```

3. Calculate the log returns and the 20-day SMA of the close prices:

```
df["log rtn"] = df["Adj Close"].apply(np.log).diff(1)
df["sma_20"] = df["Adj Close"].rolling(window=20).mean()
```

4. Create a position indicator:

```
df["position"] = (df["Adj Close"] > df["sma_20"]).astype(int)
```

Using the following snippet, we count how many times we entered a long position:

```
sum((df["position"] == 1) & (df["position"].shift(1) == 0))
```

The answer is 56.

5. Visualize the strategy over 2021:

```
fig, ax = plt.subplots(2, sharex=True)
df.loc["2021", ["Adj Close", "sma_20"]].plot(ax=ax[0])
df.loc["2021", "position"].plot(ax=ax[1])
ax[0].set_title("Preview of our strategy in 2021")
```

Executing the snippet generates the following figure:



Figure 12.1: The preview of our trading strategy based on the simple moving average

In Figure 12.1, we can clearly see how our strategy works—in the periods when the close price is above the 20-day SMA, we do have an open position. This is indicated by the value of 1 in the column containing the position information.

6. Calculate the strategy's daily and cumulative returns:

```
df["strategy rtn"] = df["position"].shift(1) * df["log rtn"]
df["strategy rtn cum"] = (
    df["strategy rtn"].cumsum().apply(np.exp)
)
```

7. Add the buy-and-hold strategy for comparison:

```
df["bh rtn cum"] = df["log rtn"].cumsum().apply(np.exp)
```

8. Plot the strategies' cumulative returns:

```
(  
    df[["bh rtn cum", "strategy rtn cum"]]  
    .plot(title="Cumulative returns")  
)
```

Executing the snippet generates the following figure:



Figure 12.2: The cumulative returns of our strategy and the buy-and-hold benchmark

In Figure 12.2, we can see the cumulative returns of both strategies. The initial conclusion could be that the simple strategy outperformed the buy-and-hold strategy over the considered time period. However, this form of a simplified backtest does not consider quite a lot of crucial aspects (for example, trading using the close price, it assumes lack of slippage and transaction costs, and so on) that can dramatically change the final outcome. In the *There's more...* section, we will see how quickly the results change when we account for transaction costs alone.

How it works...

At the very beginning, we imported the libraries and downloaded Apple's stock prices from the years 2016 to 2021. We only kept the adjusted close price for the backtest.

In *Step 3*, we calculated the log returns and the 20-day SMA. To calculate the technical indicator, we used the `rolling` method of a pandas DataFrame. However, we could have just as well used the already explored `TA-Lib` library.



We calculated the log returns, as they have a convenient property of summing up over time. If we held the position for 10 days and are interested in the final return of the position, we can simply sum up the log returns from those 10 days. For more information, please refer to *Chapter 2, Data Preprocessing*.

In *Step 4*, we created a column with information on whether we have an open position (long only) or not. As we have decided, we enter the position when the close price is above the 20-day SMA. We exit the position when the close price goes below the SMA. We have also encoded this column in the DataFrame as an integer. In *Step 5*, we plotted the close price, the 20-day SMA, and the column with the position flag. To make the plot more readable, we only plotted the data from 2021.

Step 6 is the most important one in the vectorized backtest. There, we calculated the strategy's daily and cumulative returns. To calculate the daily return, we multiplied the log return of that day with the shifted position flag. The position vector is shifted by 1 to avoid the look-ahead bias. In other words, the flag is generated using all the information up to and including time t . We can only use that information to open a position on the next trading day, that is, at time $t+1$.

An inquisitive reader might already spot another bias that is occurring with our backtest. We are correctly assuming that we can only buy on the next trading day, however, the log return is calculated as we have bought on day $t+1$ using the close price of time t , which can be very untrue depending on the market conditions. We will see how to overcome this issue with event-driven backtesting in the next recipes.

Then, we used the `cumsum` method to calculate the cumulative sum of the log returns, which corresponds to the cumulative return. Lastly, we applied the exponent function using the `apply` method.

In *Step 7*, we calculated the cumulative returns of a buy-and-hold strategy. For this one, we simply used the log returns for the calculations, skipping the step in which we multiplied the returns with the position flag.

In the last step, we plotted the cumulative returns of both strategies.

There's more...

From the initial backtest, it seems that the simple strategy is outperforming the buy-and-hold strategy. But we have also seen that over the 6 years, we have entered a long position 56 times. The total number of trades doubles, as we also exited those positions. Depending on the broker, this can result in quite significant transaction costs.

Given that transaction costs are frequently quoted in fixed percentages, we can simply calculate by how much the portfolio has changed between successive time steps, calculate the transaction costs on that basis, and then subtract them directly from our strategy's returns.

In the steps below, we show how to account for the transaction costs in a vectorized backtest. For simplicity, we assume that the transaction costs are 1%.

Execute the following steps to account for transaction costs in the vectorized backtest:

1. Calculate daily transaction costs:

```
TRANSACTION_COST = 0.01
df["tc"] = df["position"].diff(1).abs() * TRANSACTION_COST
```

In this snippet, we calculated if there is a change in our portfolio (absolute value, as we can enter or exit a position) and then multiplied that value by the transaction costs expressed as a percentage.

2. Calculate the strategy's performance accounting for transaction costs:

```
df["strategy rtn cum_tc"] = (
    (df["strategy rtn"] - df["tc"]).cumsum().apply(np.exp)
)
```

3. Plot the cumulative returns of all the strategies:

```
STRATEGY_COLS = ["bh rtn cum", "strategy rtn cum",
                  "strategy rtn cum_tc"]
(
    df
    .loc[:, STRATEGY_COLS]
    .plot(title="Cumulative returns")
)
```

Executing the snippet generates the following figure:

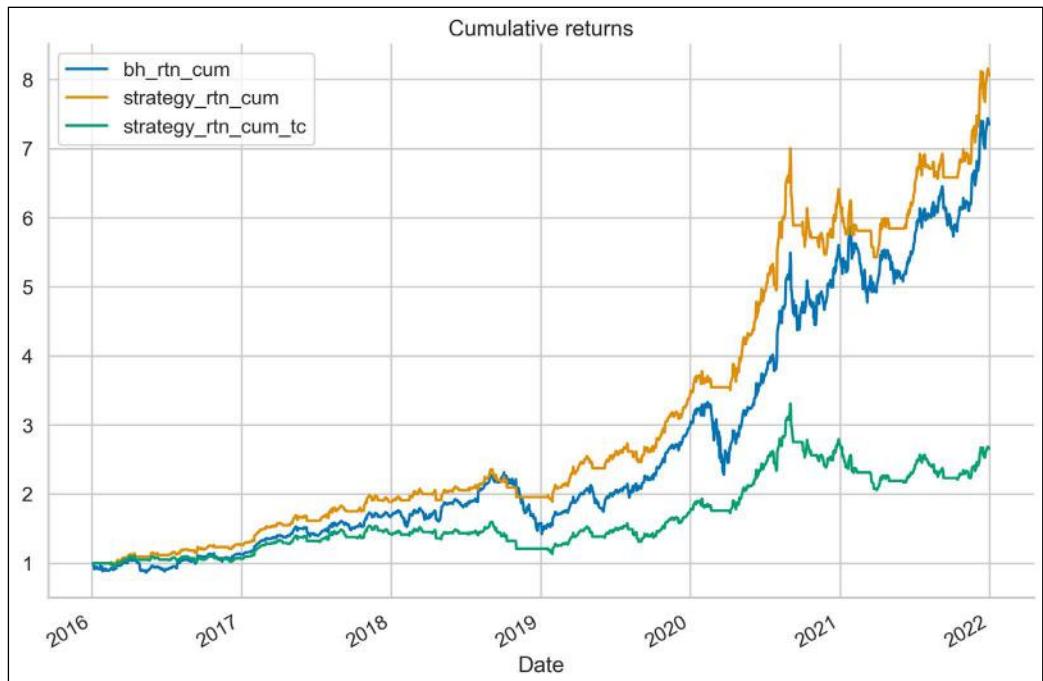


Figure 12.3: Cumulative returns of all strategies, including the one with transaction costs

After accounting for transaction costs, the performance decreased significantly and is worse than that of the buy-and-hold. And to be entirely fair, we should also account for the initial and terminal transaction costs in the buy-and-hold strategy, as we had to buy and sell the asset once.

Event-driven backtesting with backtrader

The second approach to backtesting is called event-driven backtesting. In this approach, a backtesting engine simulates the time dimension of the trading environment (you can think about it as a `for` loop going through the time and executing all the actions sequentially). This imposes more structure on the backtest, including the use of historical calendars to define when trades can actually be executed, when prices are available, and so on.

Event-driven backtesting aims to simulate all the actions and constraints encountered when executing a certain strategy while allowing for much more flexibility than the vectorized approach. For example, this approach allows for simulating potential delays in orders' execution, slippage costs, and so on. In an ideal scenario, a strategy encoded for an event-driven backtest could be easily converted into one working with live trading engines.

Nowadays, there are quite a few event-driven backtesting libraries available for Python. In this chapter, we introduce one of the most popular ones—`backtrader`. Key features of this framework include:

- A vast amount of available technical indicators (`backtrader` also provides a wrapper around the popular TA-Lib library) and performance measures.
- Ease of building and applying new indicators.
- Multiple data sources are available (including Yahoo Finance and Nasdaq Data Link), with the possibility to load external files.
- Simulating many aspects of real brokers, such as different types of orders (market, limit, and stop), slippage, commission, going long/short, and so on.
- Comprehensive and interactive visualization of the prices, TA indicators, trading signals, performance, and so on.
- Live trading with selected brokers.

For this recipe, we consider a basic strategy based on the simple moving average. As a matter of fact, it is almost identical to the one we backtested in the previous recipe using the vectorized approach. The logic of the strategy is as follows:

- When the close price becomes higher than the 20-day SMA, buy one share.
- When the close price becomes lower than the 20-day SMA and we have a share, sell it.
- We can only have a maximum of one share at any given time.
- No short selling is allowed.

We run the backtesting of this strategy using Apple's stock prices from the year 2021.

Getting ready

In this recipe (and in the rest of the chapter), we will be using two helper functions used for printing logs—`get_action_log_string` and `get_result_log_string`. Additionally, we will use a custom `MyBuySell` observer to display the position markers in different colors. You can find the definitions of those helpers in the `strategy_utils.py` file available on GitHub.

At the time of writing, the version of backtrader available at PyPI (the Python Package Index) is not the latest. Installing with a simple `pip install backtrader` command will install a version containing quite a few issues, for example, with loading the data from Yahoo Finance. To overcome this, you should install the latest version from GitHub. You can do so using the following snippet:

```
pip install git+https://github.com/momentum/backtrader.git#egg=backtrader
```

How to do it...

Execute the following steps to backtest a simple strategy using the event-driven approach:

1. Import the libraries:

```
from datetime import datetime
import backtrader as bt
from backtrader.strategies.strategy_utils import *
```

2. Download data from Yahoo Finance:

```
data = bt.feeds.YahooFinanceData(dataname="AAPL",
                                  fromdate=datetime(2021, 1, 1),
                                  todate=datetime(2021, 12, 31))
```

To make the code more readable, we first present the general outline of the class defining the trading strategy and then introduce the separate methods in the following substeps.

3. The template of the strategy is presented below:

```
class SmaStrategy(bt.Strategy):
    params = (( "ma_period", 20), )

    def __init__(self):
        # some code

    def log(self, txt):
        # some code

    def notify_order(self, order):
        # some code

    def notify_trade(self, trade):
        # some code

    def next(self):
        # some code
```

```

def start(self):
    # some code

def stop(self):
    # some code

```

- a. The `__init__` method is defined as:

```

def __init__(self):
    # keep track of close price in the series
    self.data_close = self.datas[0].close

    # keep track of pending orders
    self.order = None

    # add a simple moving average indicator
    self.sma = bt.ind.SMA(self.datas[0],
                          period=self.params.ma_period)

```

- b. The `log` method is defined as:

```

def log(self, txt):
    dt = self.datas[0].datetime.date(0).isoformat()
    print(f"{dt}: {txt}")

```

- c. The `notify_order` method is defined as:

```

def notify_order(self, order):
    if order.status in [order.Submitted, order.Accepted]:
        # order already submitted/accepted
        # no action required
        return

    # report executed order
    if order.status in [order.Completed]:

        direction = "b" if order.isbuy() else "s"
        log_str = get_action_log_string(
            dir=direction,
            action="e",
            price=order.executed.price,
            size=order.executed.size,
            cost=order.executed.value,
            commission=order.executed.comm
        )

```

```
        self.log(log_str)

        # report failed order
        elif order.status in [order.Canceled, order.Margin,
                              order.Rejected]:
            self.log("Order Failed")

        # reset order -> no pending order
        self.order = None
```

- d. The `notify_trade` method is defined as:

```
def notify_trade(self, trade):
    if not trade.isclosed:
        return

    self.log(
        get_result_log_string(
            gross=trade.pnl, net=trade.pnlcomm
        )
    )
```

- e. The `next` method is defined as:

```
def next(self):
    # do nothing if an order is pending
    if self.order:
        return

    # check if there is already a position
    if not self.position:
        # buy condition
        if self.data_close[0] > self.sma[0]:
            self.log(
                get_action_log_string(
                    "b", "c", self.data_close[0], 1
                )
            )
            self.order = self.buy()
    else:
        # sell condition
        if self.data_close[0] < self.sma[0]:
```

```

        self.log(
            get_action_log_string(
                "s", "c", self.data_close[0], 1
            )
        )
        self.order = self.sell()
    
```

- f. The `start` and `stop` methods are defined as follows:

```

def start(self):
    print(f"Initial Portfolio Value: {self.broker.get_
value():.2f}")

def stop(self):
    print(f"Final Portfolio Value: {self.broker.get_value():.2f}")

```

4. Set up the backtest:

```

cerebro = bt.Cerebro(stdstats=False)

cerebro.adddata(data)
cerebro.broker.setcash(1000.0)
cerebro.addstrategy(SmaStrategy)
cerebro.adddobserver(MyBuySell)
cerebro.adddobserver(bt.observers.Value)

```

5. Run the backtest:

```
cerebro.run()
```

Running the snippet generates the following (abbreviated) log:

```

Initial Portfolio Value: 1000.00
2021-02-01: BUY CREATED - Price: 133.15, Size: 1.00
2021-02-02: BUY EXECUTED - Price: 134.73, Size: 1.00, Cost: 134.73,
Commission: 0.00
2021-02-11: SELL CREATED - Price: 134.33, Size: 1.00
2021-02-12: SELL EXECUTED - Price: 133.56, Size: -1.00, Cost: 134.73,
Commission: 0.00
2021-02-12: OPERATION RESULT - Gross: -1.17, Net: -1.17
2021-03-16: BUY CREATED - Price: 124.83, Size: 1.00
2021-03-17: BUY EXECUTED - Price: 123.32, Size: 1.00, Cost: 123.32,
Commission: 0.00
...
2021-11-11: OPERATION RESULT - Gross: 5.39, Net: 5.39
2021-11-12: BUY CREATED - Price: 149.80, Size: 1.00

```

```
2021-11-15: BUY EXECUTED - Price: 150.18, Size: 1.00, Cost: 150.18,
Commission: 0.00
Final Portfolio Value: 1048.01
```

The log contains information about all the created and executed trades, as well as the operation results in case the position was closed.

6. Plot the results:

```
cerebro.plot(iplot=True, volume=False)
```

Running the snippet generates the following plot:



Figure 12.4: Summary of our strategy's behavior/performance over the backtested period

In *Figure 12.4*, we can see Apple's stock price, the 20-day SMA, the buy and sell orders, and the evolution of our portfolio's value over time. As we can see, this strategy made \$48 over the backtest's duration. While considering the performance, please bear in mind that the strategy is only operating with a single stock, while keeping most of the available resources in cash.

How it works...

The key idea of working with `backtrader` is that there is the main brain of the backtest—`Cerebro`—and by using different methods, we provide it with historical data, the designed trading strategy, additional metrics we want to calculate (for example, the portfolio value over the investment horizon, or the overall Sharpe ratio), information about commissions/slippage, and so on.

There are two ways of creating strategies: using signals (`bt.Signal`) or defining a full strategy (`bt.Strategy`). Both yield the same results, however, the lengthier approach (created using `bt.Strategy`) provides more logging of what is actually happening in the background. This makes it easier to debug and keep track of all operations (the level of detail included in the logging depends on our needs). That is why we start by showing that approach in this recipe.

You can find the equivalent strategy built using the signal approach in the book's GitHub repository.

After importing the libraries and helper functions in *Step 1*, we downloaded price data from Yahoo Finance using the `bt.feeds.YahooFinanceData` function.



You can also add data from a CSV file, a `pandas DataFrame`, Nasdaq Data Link, and other sources. For a list of available options, please refer to the documentation of `bt.feeds`. We show how to load data from a `pandas DataFrame` in the Notebook on GitHub.

In *Step 3*, we defined the trading strategy as a class inheriting from `bt.Strategy`. Inside the class, we defined the following methods (we were actually overwriting them to make them tailor-made for our needs):

- `__init__`: In this method, we defined the objects that we would like to keep track of. In our example, these were the close price, a placeholder for the order, and the TA indicator (SMA).
- `log`: This method is defined for logging purposes. It logs the date and the provided string. We used the helper functions `get_action_log_string` and `get_result_log_string` to create the strings with various order-related information.
- `notify_order`: This method reports the status of the order (position). In general, on day t , the indicator can suggest opening/closing a position based on the close price (assuming we are working with daily data). Then, the (market) order will be carried out on the next trading day (using the open price of time $t+1$). However, there is no guarantee that the order will be executed, as it can be canceled or we might have insufficient cash. This method also removes any pending order by setting `self.order = None`.
- `notify_trade`: This method reports the results of trades (after the positions are closed).
- `next`: This method contains the trading strategy's logic. First, we checked whether there was an order already pending, and did nothing if there was. The second check was to see whether we already had a position (enforced by our strategy, this is not a must) and if we did not, we checked whether the close price was higher than the moving average. A positive outcome resulted in an entry to the log and the placing of a buy order using `self.order = self.buy()`. This is also the place where we can choose the stake (number of assets we want to buy). The default is 1 (equivalent to using `self.buy(size=1)`).
- `start/stop`: These methods are executed at the very beginning/end of the backtest and can be used, for example, for reporting the portfolio value.

In Step 4, we set up the backtest, that is, we executed a series of operations connected to Cerebro:

- We created the instance of `bt.Cerebro` and set `stdstats=False`, in order to suppress a lot of default elements of the plot. By doing so, we avoided cluttering the output. Instead, we manually picked the interesting elements (observers and indicators).
- We added the data using the `adddata` method.
- We set up the amount of available money using the `setcash` method of the broker.
- We added the strategy using the `addstrategy` method.
- We added the observers using the `addobserver` method. We selected two observers: the custom `BuySell` observer used for displaying the buy/sell decisions on the plot (denoted by green and red triangles), and the `Value` observer used for tracking the evolution of the portfolio's value over time.

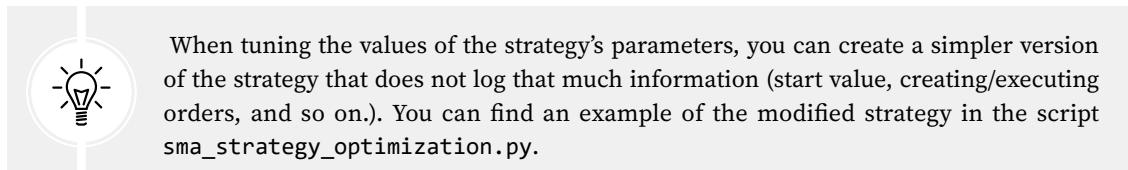
The last steps involved running the backtest with `cerebro.run()` and plotting the results with `cerebro.plot()`. In the latter step, we disabled displaying the volume charts to avoid cluttering the graph.

Some additional points about backtesting with `backtrader`:

- By design, `Cerebro` should only be used once. If we want to run another backtest, we should create a new instance, not add something to it after starting the calculations.
- In general, a strategy built using `bt.Signal` uses only one signal. However, we can combine multiple signals based on different conditions by using `bt.SignalStrategy` instead.
- When we do not specify otherwise, all orders are placed for one unit of the asset.
- `backtrader` automatically handles the warm-up period. In this case, no trade can be carried out until there are enough data points to calculate the 20-day SMA. When considering multiple indicators at once, `backtrader` automatically selects the longest necessary period.

There's more...

It is worth mentioning that `backtrader` has parameter optimization capabilities, which we present in the code that follows. The code is a modified version of the strategy from this recipe, in which we optimize the number of days used for calculating the SMA.



The following list provides details of modifications to the code (we only show the relevant ones, as the bulk of the code is identical to the code used before):

- Instead of using `cerebro.addstrategy`, we use `cerebro.optstrategy`, and provide the defined strategy object and the range of parameter values:

```
cerebro.optstrategy(SmaStrategy, ma_period=range(10, 31))
```

- We modify the `stop` method to also log the considered value of the `ma_period` parameter.
- We increase the number of CPU cores when running the extended backtesting:

```
cerebro.run(maxcpus=4)
```

We present the results in the following summary (please bear in mind that the order of parameters can be shuffled when using multiple cores):

```
2021-12-30: (ma_period = 10) --- Terminal Value: 1018.82
2021-12-30: (ma_period = 11) --- Terminal Value: 1022.45
2021-12-30: (ma_period = 12) --- Terminal Value: 1022.96
2021-12-30: (ma_period = 13) --- Terminal Value: 1032.44
2021-12-30: (ma_period = 14) --- Terminal Value: 1027.37
2021-12-30: (ma_period = 15) --- Terminal Value: 1030.53
2021-12-30: (ma_period = 16) --- Terminal Value: 1033.03
2021-12-30: (ma_period = 17) --- Terminal Value: 1038.95
2021-12-30: (ma_period = 18) --- Terminal Value: 1043.48
2021-12-30: (ma_period = 19) --- Terminal Value: 1046.68
2021-12-30: (ma_period = 20) --- Terminal Value: 1048.01
2021-12-30: (ma_period = 21) --- Terminal Value: 1044.00
2021-12-30: (ma_period = 22) --- Terminal Value: 1046.98
2021-12-30: (ma_period = 23) --- Terminal Value: 1048.62
2021-12-30: (ma_period = 24) --- Terminal Value: 1051.08
2021-12-30: (ma_period = 25) --- Terminal Value: 1052.44
2021-12-30: (ma_period = 26) --- Terminal Value: 1051.30
2021-12-30: (ma_period = 27) --- Terminal Value: 1054.78
2021-12-30: (ma_period = 28) --- Terminal Value: 1052.75
2021-12-30: (ma_period = 29) --- Terminal Value: 1045.74
2021-12-30: (ma_period = 30) --- Terminal Value: 1047.60
```

We see that the strategy performed best when we used 27 days for calculating the SMA.



We should always keep in mind that tuning the hyperparameters of a strategy comes together with a higher risk of overfitting!

See also

You can refer to the following book for more information about algorithmic trading and building successful trading strategies:

- Chan, E. (2013). *Algorithmic Trading: Winning Strategies and Their Rationale* (Vol. 625). John Wiley & Sons.

Backtesting a long/short strategy based on the RSI

The relative strength index (RSI) is an indicator that uses the closing prices of an asset to identify oversold/overbought conditions. Most commonly, the RSI is calculated using a 14-day period, and it is measured on a scale from 0 to 100 (it is an oscillator). Traders usually buy an asset when it is oversold (if the RSI is below 30), and sell when it is overbought (if the RSI is above 70). More extreme high/low levels, such as 80-20, are used less frequently and, at the same time, imply stronger momentum.

In this recipe, we build a trading strategy with the following rules:

- We can go long and short.
- For calculating the RSI, we use 14 periods (trading days).
- Enter a long position if the RSI crosses the lower threshold (standard value of 30) upward; exit the position when the RSI becomes larger than the middle level (value of 50).
- Enter a short position if the RSI crosses the upper threshold (standard value of 70) downward; exit the position when the RSI becomes smaller than 50.
- Only one position can be open at a time.

We evaluate the strategy on Meta's stock in 2021 and apply a commission of 0.1%.

How to do it...

Execute the following steps to implement and backtest a strategy based on the RSI:

1. Import the libraries:

```
from datetime import datetime
import backtrader as bt
from backtrader_strategies.strategy_utils import *
```

2. Define the signal strategy based on `bt.SignalStrategy`:

```
class RsiSignalStrategy(bt.SignalStrategy):
    params = dict(rsi_periods=14, rsi_upper=70,
                  rsi_lower=30, rsi_mid=50)

    def __init__(self):
        # add RSI indicator
        rsi = bt.indicators.RSI(period=self.p.rsi_periods,
                               upperband=self.p.rsi_upper,
                               lowerband=self.p.rsi_lower)
        # add RSI from TA-lib just for reference
        bt.talib.RSI(self.data, plotname="TA_RSI")

        # Long condition (with exit)
        rsi_signal_long = bt.ind.CrossUp(
```

```
        rsi, self.p.rsi_lower, plot=False
    )
    self.signal_add(bt.SIGNAL_LONG, rsi_signal_long)
    self.signal_add(
        bt.SIGNAL_LONGEXIT, -(rsi > self.p.rsi_mid)
    )

    # short condition (with exit)
    rsi_signal_short = -bt.ind.CrossDown(
        rsi, self.p.rsi_upper, plot=False
    )
    self.signal_add(bt.SIGNAL_SHORT, rsi_signal_short)
    self.signal_add(
        bt.SIGNAL_SHORTEXIT, rsi < self.p.rsi_mid
    )
```

3. Download data:

```
data = bt.feeds.YahooFinanceData(dataname="META",
                                  fromdate=datetime(2021, 1, 1),
                                  todate=datetime(2021, 12, 31))
```

4. Set up and run the backtest:

```
cerebro = bt.Cerebro(stdstats=False)

cerebro.addstrategy(RsiSignalStrategy)
cerebro.adddata(data)
cerebro.addsizer(bt.sizersSizerFix, stake=1)
cerebro.broker.setcash(1000.0)
cerebro.broker.setcommission(commission=0.001)
cerebro.addobserver(MyBuySell)
cerebro.addobserver(bt.observers.Value)

print(
    f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}"
)
cerebro.run()
print(
    f"Final Portfolio Value: {cerebro.broker.getvalue():.2f}"
)
```

After running the snippet, we see the following output:

```
Starting Portfolio Value: 1000.00
Final Portfolio Value: 1042.56
```

5. Plot the results:

```
cerebro.plot(iplot=True, volume=False)
```

Running the snippet generates the following plot:



Figure 12.5: Summary of our strategy's behavior/performance over the backtested period

We look at the triangles in pairs. The first triangle in a pair indicates opening a position (going long if the triangle is green and facing up; going short if the triangle is red and facing down). The next triangle in the opposite direction indicates closing of a position. We can match the opening and closing of positions with the RSI located in the lower part of the chart. Sometimes, there are multiple triangles of the same color in sequence. That is because the RSI fluctuates around the line of opening a position, crossing it multiple times. But the actual position is only opened on the first instance of a signal (no accumulation is the default setting of all backtests).

How it works...

In this recipe, we presented the second approach to defining strategies in backtrader, that is, using signals. A signal is represented as a number, for example, the difference between the current data point and some TA indicator. If the signal is positive, it is an indication to go long (buy). A negative one is an indication to take a short position (sell). The value of 0 means there is no signal.

After importing the libraries and the helper functions, we defined the trading strategy using `bt.SignalStrategy`. As this is a strategy involving multiple signals (various entry/exit conditions), we had to use `bt.SignalStrategy` instead of simply `bt.Signal`. First, we defined the indicator (RSI), with selected arguments. We also added the second instance of the RSI indicator, just to show that `backtrader` provides an easy way to use indicators from the popular TA-Lib library (the library must be installed for the code to work). The trading strategy does not depend on this second indicator—it is only plotted for reference. In general, we could add an arbitrary number of indicators.



Even when adding indicators for reference only, their existence influences the “warm-up period.” For example, if we additionally included a 200-day SMA indicator, no trade would be carried out before there exists at least one value for the SMA indicator.

The next step was to define signals. To do so, we used the `bt.CrossUp/bt.CrossDown` indicators, which returned 1 if the first series (price) crossed the second (upper or lower RSI threshold) from below/above, respectively. For entering a short position, we made the signal negative, by adding a minus in front of the `bt.CrossDown` indicator.



We can disable printing any indicator, by adding `plot=False` to the function call.

The following is a description of the available signal types:

- `LONGSHORT`: This type takes into account both long and short indications from the signal.
- `LONG`: Positive signals indicate going long; negative ones are used for closing the long position.
- `SHORT`: Negative signals indicate going short; positive ones are used for closing the short position.
- `LONGEXIT`: A negative signal is used to exit a long position.
- `SHORTEXIT`: A positive signal is used to exit a short position.

Exiting positions can be more complex, which in turn enables users to build more sophisticated strategies. We describe the logic below:

- `LONG`: If there is a `LONGEXIT` signal, it is used for exiting the long position, instead of the behavior mentioned above. If there is a `SHORT` signal and no `LONGEXIT` signal, the `SHORT` signal is used to close the long position before opening a short one.
- `SHORT`: If there is a `SHORTEXIT` signal, it is used for exiting the short position, instead of the behavior mentioned above. If there is a `LONG` signal and no `SHORTEXIT` signal, the `LONG` signal is used to close the short position before opening a long one.



As you might have already realized, the signal is calculated for every time point (as visualized at the bottom of the plot), which effectively creates a continuous stream of positions to be opened/closed (the signal value of 0 is not very likely to happen). That is why, by default, backtrader disables accumulation (the constant opening of new positions, even when we have one already opened) and concurrency (generating new orders without hearing back from the broker whether the previously submitted ones were executed successfully).

As the last step of defining the strategy, we added tracking of all the signals, by using the `signal_add` method. For exiting the positions, the conditions we used (RSI value higher/lower than 50) resulted in a Boolean, which we had to negate when exiting a long position: in Python, `-True` has the same meaning as `-1`.

In Step 3, we downloaded Meta's stock prices from 2021.

Then, we set up the backtest. Most of the steps should already be familiar, that is why we focus only on the new ones:

- Adding a sizer using the `addsize` method—we did not have to do it at this point, as by default, backtrader uses the stake of 1, that is, 1 unit of the asset will be purchased/sold. However, we wanted to show at which point we can modify the order size when creating a trading strategy using the signal approach.
- Setting up the commission to 0.1% using the `setcommission` method of the broker.
- We also accessed and printed the portfolio's current value before and after running the backtest. To do so, we used the `getvalue` method of broker.

In the very last step, we plotted the results of the backtest.

There's more...

In this recipe, we have introduced a couple of new concepts to the backtesting framework—sizers and commission. There are a few more useful things we can experiment with using those two components.

Going “all-in”

Before, our simple strategy only went long or short with a single unit of the asset. However, we can easily modify this behavior to use all the available cash. We simply add the `AllInSizer` sizer using the `addsize` method:

```
cerebro = bt.Cerebro(stdstats=False)

cerebro.addstrategy(RsiSignalStrategy)
cerebro.adddata(data)
cerebro.addsize(bt.sizers.AllInSizer)
cerebro.broker.setcash(1000.0)
cerebro.broker.setcommission(commission=0.001)
cerebro.addobserver(bt.observers.Value)
```

```
print(f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}")
cerebro.run()
print(f"Final Portfolio Value: {cerebro.broker.getvalue():.2f}")
```

Running the backtest generates the following result:

```
Starting Portfolio Value: 1000.00
Final Portfolio Value: 1183.95
```

The result is clearly better than what we achieved using only a single unit at a time.

Fixed commission per share

In our initial backtest of the RSI-based strategy, we used a 0.1% commission fee. However, some brokers might have a different commission scheme, for example, a fixed commission per share.

To incorporate such information, we need to define a custom class storing the commission scheme. We can inherit from `bt.CommInfoBase` and add the required information:

```
class FixedCommissionShare(bt.CommInfoBase):
    """
    Scheme with fixed commission per share
    """

    params = (
        ("commission", 0.03),
        ("stocklike", True),
        ("commtype", bt.CommInfoBase.COMM_FIXED),
    )

    def _getcommission(self, size, price, pseudoexec):
        return abs(size) * self.p.commission
```

The most important aspects of the definition are the fixed commission of \$0.03 per share and the way that the commission is calculated in the `_getcommission` method. We take the absolute value of the size and multiply it by the fixed commission.

We can then easily input that information into the backtest. Building on top of the previous example with the “all-in” strategy, the code would look as follows:

```
cerebro = bt.Cerebro(stdstats=False)

cerebro.addstrategy(RsiSignalStrategy)
cerebro.adddata(data)
cerebro.addsizer(bt.sizers.AllInSizer)
```

```
cerebro.broker.setcash(1000.0)
cerebro.broker.addcommissioninfo(FixedCommissionShare())
cerebro.addobserver(bt.observers.Value)

print(f"Starting Portfolio Value: {cerebro.broker.getvalue():.2f}")
cerebro.run()
print(f"Final Portfolio Value: {cerebro.broker.getvalue():.2f}")
```

With the following result:

```
Starting Portfolio Value: 1000.00
Final Portfolio Value: 1189.94
```

These numbers lead to the conclusion that the 0.01% commission was actually higher than 3 cents per share.

Fixed commission per order

Other brokers might offer a fixed commission per order. In the following snippet, we define a custom commission scheme in which we pay \$2.5 per order, regardless of its size.

We changed the value of the `commission` parameter and the way commission is calculated in the `_getcommission` method. This time, this method always returns the \$2.5 we specified before:

```
class FixedCommissionOrder(bt.CommInfoBase):
    """
    Scheme with fixed commission per order
    """

    params = (
        ("commission", 2.5),
        ("stocklike", True),
        ("commtype", bt.CommInfoBase.COMM_FIXED),
    )

    def _getcommission(self, size, price, pseudoexec):
        return self.p.commission
```

We do not include the backtest setup, as it would be almost identical to the previous one. We only need to pass a different class using the `addcommissioninfo` method. The result of the backtest is:

```
Starting Portfolio Value: 1000.00
Final Portfolio Value: 1174.70
```

See also

Below, you might find useful references to backtrader's documentation:

- To read more about sizers: <https://www.backtrader.com/docu/sizers-reference/>
- To read more about commission schemes and the available parameters: <https://www.backtrader.com/docu/commission-schemes/commission-schemes/>

Backtesting a buy/sell strategy based on Bollinger bands

Bollinger bands are a statistical method, used for deriving information about the prices and volatility of a certain asset over time. To obtain the Bollinger bands, we need to calculate the moving average and standard deviation of the time series (prices), using a specified window (typically 20 days). Then, we set the upper/lower bands at K times (typically 2) the moving standard deviation above/below the moving average.

The interpretation of the bands is quite simple: the bands widen with an increase in volatility and contract with a decrease in volatility.

In this recipe, we build a simple trading strategy that uses Bollinger bands to identify underbought and oversold levels and then trade based on those areas. The rules of the strategy are as follows:

- Buy when the price crosses the lower Bollinger band upward.
- Sell (only if stocks are in possession) when the price crosses the upper Bollinger band downward.
- All-in strategy—when creating a buy order, buy as many shares as possible.
- Short selling is not allowed.

We evaluate the strategy on Microsoft's stock in 2021. Additionally, we set the commission to be equal to 0.1%.

How to do it...

Execute the following steps to implement and backtest a strategy based on the Bollinger bands:

1. Import the libraries:

```
import backtrader as bt
import datetime
import pandas as pd
from backtrader_strategies.strategy_utils import *
```

To make the code more readable, we first present the general outline of the class defining the trading strategy and then introduce the separate methods in the following substeps.

2. Define the strategy based on the Bollinger bands:

```
class BollingerBandStrategy(bt.Strategy):  
    params = (({"period", 20},  
              {"devfactor", 2.0}),)  
    def __init__(self):  
        # some code  
  
    def log(self, txt):  
        # some code  
  
    def notify_order(self, order):  
        # some code  
  
    def notify_trade(self, trade):  
        # some code  
  
    def next_open(self):  
        # some code  
  
    def start(self):  
        print(f"Initial Portfolio Value: {self.broker.get_value():.2f}")  
  
    def stop(self):  
        print(f"Final Portfolio Value: {self.broker.get_value():.2f}")
```

When defining strategies using the strategy approach, there is quite some boilerplate code. That is why in the following substeps, we only mention the methods that are different from the ones we have previously explained. You can also find the strategy's entire code in the book's GitHub repository:

- a. The `__init__` method is defined as:

```
def __init__(self):  
    # keep track of prices  
    self.data_close = self.datas[0].close  
    self.data_open = self.datas[0].open  
  
    # keep track of pending orders  
    self.order = None  
  
    # add Bollinger Bands indicator and track buy/sell  
    # signals
```

```

        self.b_band = bt.ind.BollingerBands(
            self.datas[0],
            period=self.p.period,
            devfactor=self.p.devfactor
        )
        self.buy_signal = bt.ind.Crossover(
            self.datas[0],
            self.b_band.lines.bot,
            plotname="buy_signal"
        )
        self.sell_signal = bt.ind.Crossover(
            self.datas[0],
            self.b_band.lines.top,
            plotname="sell_signal"
        )
    )

```

- b. The `next_open` method is defined as:

```

def next_open(self):
    if not self.position:
        if self.buy_signal > 0:
            # calculate the max number of shares ("all-in")
            size = int(
                self.broker.getcash() / self.datas[0].open
            )
            # buy order
            log_str = get_action_log_string(
                "b", "c",
                price=self.data_close[0],
                size=size,
                cash=self.broker.getcash(),
                open=self.data_open[0],
                close=self.data_close[0]
            )
            self.log(log_str)
            self.order = self.buy(size=size)
    else:
        if self.sell_signal < 0:
            # sell order
            log_str = get_action_log_string(
                "s", "c", self.data_close[0],
                self.position.size

```

```
)  
self.log(log_str)  
self.order = self.sell(size=self.position.size)
```

3. Download data:

```
data = bt.feeds.YahooFinanceData(  
    dataname="MSFT",  
    fromdate=datetime.datetime(2021, 1, 1),  
    todate=datetime.datetime(2021, 12, 31)  
)
```

4. Set up the backtest:

```
cerebro = bt.Cerebro(stdstats=False, cheat_on_open=True)  
  
cerebro.addstrategy(BollingerBandStrategy)  
cerebro.adddata(data)  
cerebro.broker.setcash(10000.0)  
cerebro.broker.setcommission(commission=0.001)  
cerebro.addobserver(MyBuySell)  
cerebro.addobserver(bt.observers.Value)  
cerebro.addanalyzer(  
    bt.analyzers.Returns, _name="returns"  
)  
cerebro.addanalyzer(  
    bt.analyzers.TimeReturn, _name="time_return"  
)
```

5. Run the backtest:

```
backtest_result = cerebro.run()
```

Running the backtest generates the following (abbreviated) log:

```
Initial Portfolio Value: 10000.00  
2021-03-01: BUY CREATED - Price: 235.03, Size: 42.00, Cash: 10000.00,  
Open: 233.99, Close: 235.03  
2021-03-01: BUY EXECUTED - Price: 233.99, Size: 42.00, Cost: 9827.58,  
Commission: 9.83  
2021-04-13: SELL CREATED - Price: 256.40, Size: 42.00  
2021-04-13: SELL EXECUTED - Price: 255.18, Size: -42.00, Cost:  
9827.58, Commission: 10.72  
2021-04-13: OPERATION RESULT - Gross: 889.98, Net: 869.43
```

```

...
2021-12-07: BUY CREATED - Price: 334.23, Size: 37.00, Cash: 12397.10,
Open: 330.96, Close: 334.23
2021-12-07: BUY EXECUTED - Price: 330.96, Size: 37.00, Cost: 12245.52,
Commission: 12.25
Final Portfolio Value: 12668.27

```

6. Plot the results:

```
cerebro.plot(iplot=True, volume=False)
```

Running the snippet generates the following plot:

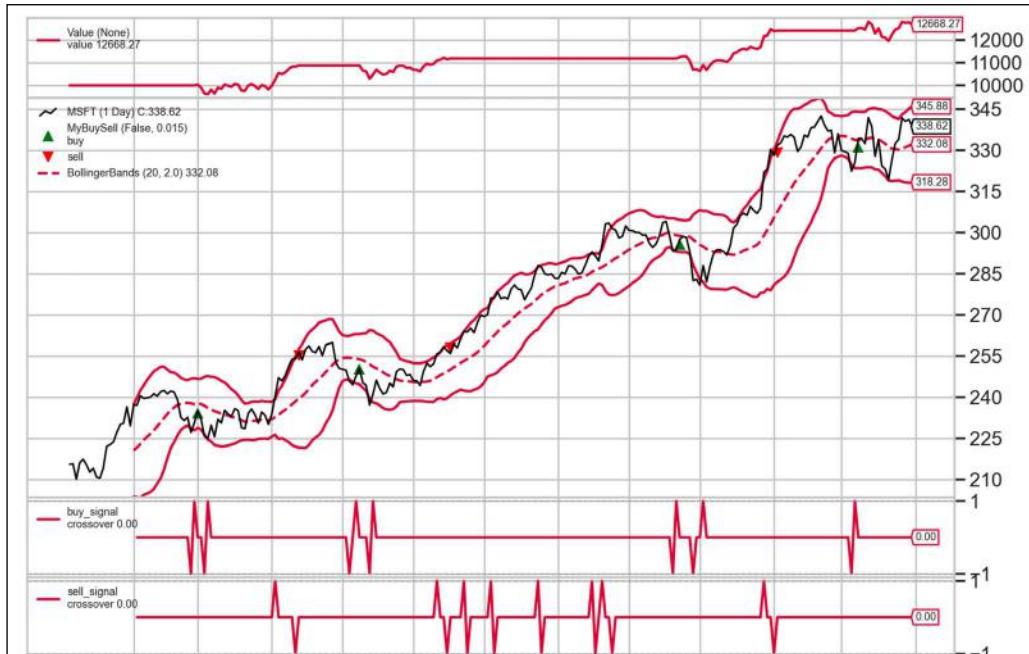


Figure 12.6: Summary of our strategy's behavior/performance over the backtested period

We can see that the strategy managed to make money, even after accounting for commission costs. The flat periods in the portfolio's value represent periods when we did not have an open position.

7. Investigate different returns metrics:

```
backtest_result[0].analyzers.returns.get_analysis()
```

Running the code generates the following output:

```

OrderedDict([('rtot', 0.2365156915893157),
             ('ravg', 0.0009422935919893056),
             ('rnorm', 0.2680217199688534),
             ('rnorm100', 26.80217199688534)])

```

8. Extract daily portfolio returns and plot them:

```
returns_dict = (
    backtest_result[0].analyzers.time_return.get_analysis()
)
returns_df = (
    pd.DataFrame(list(returns_dict.items()),
                columns = ["date", "return"])
    .set_index("date")
)
returns_df.plot(title="Strategy's daily returns")
```



Figure 12.7: Daily portfolio returns of the strategy based on Bollinger bands

We can see that the flat periods in the portfolio's returns in *Figure 12.7* correspond to the periods during which we had no open positions, as can be seen in *Figure 12.6*.

How it works...

There are a lot of similarities between the code used for creating the Bollinger bands-based strategy and that used in the previous recipes. That is why we only discuss the novelties and refer you to the *Event-driven backtesting with backtrader* recipe for more details.

As we were going all-in in this strategy, we had to use a method called `cheat_on_open`. This means that we calculated the signals using day t 's close price, but calculated the number of shares we wanted to buy based on day $t+1$'s open price. To do so, we had to set `cheat_on_open=True` when instantiating the `Cerebro` object.

As a result, we also defined a `next_open` method instead of `next` within the `Strategy` class. This clearly indicated to `Cerebro` that we were cheating on open. Before creating a potential buy order, we manually calculated the maximum number of shares we could buy using the open price from day $t+1$.

When calculating the buy/sell signals based on the Bollinger bands, we used the `CrossOver` indicator. It returned the following:

- 1 if the first data (price) crossed the second data (indicator) upward
- -1 if the first data (price) crossed the second data (indicator) downward



We can also use `CrossUp` and `CrossDown` functions when we want to consider crossing from only one direction. The buy signal would look like this:
`self.buy_signal = bt.ind.CrossUp(self.datas[0], self.b_band.lines.bot).`

The last addition included utilizing analyzers—`backtrader` objects that help to evaluate what is happening with the portfolio. In this recipe, we used two analyzers:

- **Returns:** A collection of different logarithmic returns, calculated over the entire timeframe: total compound return, the average return over the entire period, and the annualized return.
- **TimeReturn:** A collection of returns over time (using a provided timeframe, in this case, daily data).



We can obtain the same result as from the `TimeReturn` analyzer by adding an observer with the same name: `cerebro.addobserver(bt.observers.TimeReturn)`. The only difference is that the observer will be plotted on the main results plot, which is not always desired.

There's more...

We have already seen how to extract the daily returns from the backtest. This creates a perfect opportunity to combine that information with the functionalities of the `quantstats` library. Using the following snippet, we can calculate a variety of metrics to evaluate our portfolio's performance in detail. Additionally, we compare the performance of our strategy to a simple buy-and-hold strategy (which, for simplicity, does not include the transaction costs):

```
import quantstats as qs
qs.reports.metrics(returns_df,
                    benchmark="MSFT",
                    mode="basic")
```

Running the snippet generates the following report:

	Strategy	Benchmark
Start Period	2021-01-04	2021-01-04
End Period	2021-12-30	2021-12-30
Risk-Free Rate	0.0%	0.0%
Time in Market	42.0%	100.0%
 Cumulative Return	26.68%	57.18%
CAGR%	27.1%	58.17%
 Sharpe	1.65	2.27
Sortino	2.68	3.63
Sortino/V2	1.9	2.57
Omega	1.52	1.52

For brevity's sake, we only present the few main pieces of information available in the report.



In *Chapter 11, Asset Allocation*, we mentioned that an alternative library to `quantstats` is `pyfolio`. The latter has the potential disadvantage of not being actively maintained anymore. However, `pyfolio` is nicely integrated with `backtrader`. We can easily add a dedicated analyzer (`bt.analyzers.PyFolio`). For an example of implementation, please see the book's GitHub repository.

Backtesting a moving average crossover strategy using crypto data

So far, we have created and backtested a few strategies on stocks. In this recipe, we cover another popular asset class—cryptocurrencies. There are a few key differences in handling crypto data:

- Cryptos can be traded 24/7
- Cryptos can be traded using fractional units

As we want our backtests to closely resemble real-life trading, we should account for those crypto-specific characteristics in our backtests. Fortunately, the `backtrader` framework is very flexible and we can slightly adjust our already-established approach to handle this new asset class.



Some brokers also allow for buying fractional shares of stocks.

In this recipe, we backtest a moving average crossover strategy with the following rules:

- We are only interested in Bitcoin and use daily data from 2021.
- We use two moving averages with window sizes of 20-days (fast one) and 50-days (slow one).
- If the fast MA crosses over the slow one upward, we allocate 70% of available cash to buying BTC.
- If the fast MA crosses over the slow one downward, we sell all the BTC we have.
- Short selling is not allowed.

How to do it...

Execute the following steps to implement and backtest a strategy based on the moving average crossover:

1. Import the libraries:

```
import backtrader as bt
import datetime
import pandas as pd
from backtrader_strategies.strategy_utils import *
```

2. Define the commission scheme allowing for fractional trades:

```
class FractionalTradesCommission(bt.CommissionInfo):
    def getsize(self, price, cash):
        """Returns the fractional size"""
        return self.p.leverage * (cash / price)
```

To make the code more readable, we first present the general outline of the class defining the trading strategy and then introduce the separate methods in the following substeps.

3. Define the SMA crossover strategy:

```
class SMACrossoverStrategy(bt.Strategy):
    params = (
        ("ma_fast", 20),
        ("ma_slow", 50),
        ("target_perc", 0.7)
    )

    def __init__(self):
        # some code

    def log(self, txt):
        # some code
```

```
def notify_order(self, order):
    # some code

def notify_trade(self, trade):
    # some code

def next(self):
    # some code

def start(self):
    print(f"Initial Portfolio Value: {self.broker.get_value():.2f}")

def stop(self):
    print(f"Final Portfolio Value: {self.broker.get_value():.2f}")
```

- a. The `__init__` method is defined as:

```
def __init__(self):
    # keep track of close price in the series
    self.data_close = self.datas[0].close

    # keep track of pending orders
    self.order = None

    # calculate the SMAs and get the crossover signal
    self.fast_ma = bt.indicators.MovingAverageSimple(
        self.datas[0],
        period=self.params.ma_fast
    )
    self.slow_ma = bt.indicators.MovingAverageSimple(
        self.datas[0],
        period=self.params.ma_slow
    )
    self.ma_crossover = bt.indicators.CrossOver(self.fast_ma,
                                                self.slow_ma)
```

- b. The `next` method is defined as:

```
def next(self):

    if self.order:
        # pending order execution. Waiting in orderbook
        return
```

```
if not self.position:  
    if self.ma_crossover > 0:  
        self.order = self.order_target_percent(  
            target=self.params.target_perc  
        )  
        log_str = get_action_log_string(  
            "b", "c",  
            price=self.data_close[0],  
            size=self.order.size,  
            cash=self.broker.getcash(),  
            open=self.data_open[0],  
            close=self.data_close[0]  
        )  
        self.log(log_str)  
  
    else:  
        if self.ma_crossover < 0:  
            # sell order  
            log_str = get_action_log_string(  
                "s", "c", self.data_close[0],  
                self.position.size  
            )  
            self.log(log_str)  
            self.order = (  
                self.order_target_percent(target=0)  
            )
```

4. Download the BTC-USD data:

```
data = bt.feeds.YahooFinanceData(  
    dataname="BTC-USD",  
    fromdate=datetime.datetime(2020, 1, 1),  
    todate=datetime.datetime(2021, 12, 31)  
)
```

5. Set up the backtest:

```
cerebro = bt.Cerebro(stdstats=False)

cerebro.addstrategy(SMACrossoverStrategy)
cerebro.adddata(data)
cerebro.broker.setcash(10000.0)
cerebro.broker.addcommissioninfo(
    FractionalTradesCommission(commission=0.001)
)
cerebro.addobserver(MyBuySell)
cerebro.addobserver(bt.observers.Value)
cerebro.addanalyzer(
    bt.analyzers.TimeReturn, _name="time_return"
)
```

6. Run the backtest:

```
backtest_result = cerebro.run()
```

Running the snippet generates the following (abbreviated) log:

```
Initial Portfolio Value: 10000.00
2020-04-19: BUY CREATED - Price: 7189.42, Size: 0.97, Cash: 10000.00,
Open: 7260.92, Close: 7189.42
2020-04-20: BUY EXECUTED - Price: 7186.87, Size: 0.97, Cost: 6997.52,
Commission: 7.00
2020-06-29: SELL CREATED - Price: 9190.85, Size: 0.97
2020-06-30: SELL EXECUTED - Price: 9185.58, Size: -0.97, Cost: 6997.52,
Commission: 8.94
2020-06-30: OPERATION RESULT - Gross: 1946.05, Net: 1930.11
...
Final Portfolio Value: 43547.99
```

In the excerpt from the full log, we can see that we are now operating with fractional positions. Also, the strategy has generated quite significant returns—we have approximately quadrupled the initial portfolio's value.

7. Plot the results:

```
cerebro.plot(iplot=True, volume=False)
```

Running the snippet generates the following plot:



Figure 12.8: Summary of our strategy's behavior/performance over the backtested period

We have already established that we have generated >300% returns using our strategy. However, we can also see in *Figure 12.8* that the great performance might simply be due to the gigantic increase in BTC's price over the considered period.

Using code identical to the code used in the previous recipe, we can compare the performance of our strategy to the simple buy-and-hold strategy. This way, we can verify how our active strategy performed compared to a static benchmark. We present the abbreviated performance comparison below, while the code can be found in the book's GitHub repository.

	Strategy	Benchmark
Start Period	2020-01-01	2020-01-01
End Period	2021-12-30	2021-12-30
Risk-Free Rate	0.0%	0.0%
Time in Market	57.0%	100.0%
Cumulative Return	335.48%	555.24%
CAGR%	108.89%	156.31%
Sharpe	1.6	1.35
Sortino	2.63	1.97

Sortino/V2	1.86	1.4
Omega	1.46	1.46

Unfortunately, our strategy did not outperform the benchmark over the analyzed timeframe. This confirms our initial suspicion that the good performance is connected to the increase in BTC's price over the considered period.

How it works...

After importing the libraries, we defined a custom commission scheme in order to allow for fractional shares. Before, when we created a custom commission scheme, we inherited from `bt.CommInfoBase` and we modified the `_getcommission` method. This time, we inherited from `bt.CommissionInfo` and modified the `getsize` method to return a fractional value depending on the available cash and the asset's price.

In *Step 3* (and its substeps) we defined the moving average crossover strategy. By this recipe, most of the code will already look very familiar. A new thing we have applied here is the different type of order, that is, `order_target_percent`. Using this type of order indicates that we want the given asset to be X% of our portfolio.

It is a very convenient method because we leave the exact order size calculations to `backtrader`. If, at the moment of issuing the order, we are below the specified target percentage, we will buy more of the asset. If we are above it, we will sell some amount of the asset.

For exiting the position, we indicate that we want BTC to be 0% of our portfolio, which is equivalent to selling all we have. By using `order_target_percent` with the target of zero, we do not have to track/access the current number of units we possess.

In *Step 4*, we downloaded the daily BTC prices (in USD) from 2021. In the following steps, we set up the backtest, ran it, and plotted the results. The only thing worth mentioning is that we had to add the custom commission scheme (containing the fractional share logic) using the `addcommissioninfo` method.

There's more...

In the recipe, we have introduced the target order. `backtrader` offers three types of target orders:

- `order_target_percent`: Indicates the percentage of the current portfolio's value we want to have in the given asset
- `order_target_size`: Indicates the target number of units of a given asset we want to have in the portfolio
- `order_target_value`: Indicates the asset's target value in monetary units that we want to have in the portfolio

Target orders are very useful when we know the target percentage/value/size of a given asset, but do not want to spend additional time calculating whether we should buy additional units or sell them to arrive at the target.

There is also one more important thing to mention about fractional shares. In this recipe, we have defined a custom commission scheme that accounts for the fractional shares and then we used the target orders to buy/sell the asset. This way, when the engine was calculating the number of units to trade in order to arrive at the target, it knew it could use fractional values.

However, there is another way of using fractional shares without defining a custom commission scheme. We simply need to manually calculate the number of shares we want to buy/sell and create an order with a given stake. We did something very similar in the previous recipe, but there, we rounded the potential fractional values to an integer. For an implementation of the SMA crossover strategy with manual fractional order size calculations, please refer to the book's GitHub repository.

Backtesting a mean-variance portfolio optimization

In the previous chapter, we covered asset allocation and mean-variance optimization. Combining mean-variance optimization with a backtest would be an interesting exercise, especially because it involves working with multiple assets at once.

In this recipe, we backtest the following allocation strategy:

- We consider the FAANG stocks.
- Every Friday after the market closes, we find the tangency portfolio (maximizing the Sharpe ratio). Then, we create target orders to match the calculated optimal weights on Monday when the market opens.
- We assume we need to have at least 252 data points to calculate the expected returns and the covariance matrix (using the Ledoit-Wolf approach).

For this exercise, we download the prices of the FAANG stocks from 2020 to 2021. Due to the warm-up period we set up for calculating the weights, the trading actually happens only in 2021.

Getting ready

As we will be working with fractional shares in this recipe, we need to use the custom commission scheme (`FractionalTradesCommission`) defined in the previous recipe.

How to do it...

Execute the following steps to implement and backtest a strategy based on the mean-variance portfolio optimization:

1. Import the libraries:

```
from datetime import datetime
import backtrader as bt
import pandas as pd
from pypfopt.expected_returns import mean_historical_return
from pypfopt.risk_models import CovarianceShrinkage
from pypfopt.efficient_frontier import EfficientFrontier
from backtrader_strategies.strategy_utils import *
```

To make the code more readable, we first present the general outline of the class defining the trading strategy and then introduce the separate methods in the following substeps.

2. Define the strategy:

```
class MeanVariancePortfStrategy(bt.Strategy):
    params = ((n_periods, 252), )

    def __init__(self):
        # track number of days
        self.day_counter = 0

    def log(self, txt):
        dt = self.datas[0].datetime.date(0).isoformat()
        print(f"{dt}: {txt}")

    def notify_order(self, order):
        # some code

    def notify_trade(self, trade):
        # some code

    def next(self):
        # some code

    def start(self):
        print(f"Initial Portfolio Value: {self.broker.getvalue():.2f}")

    def stop(self):
        print(f"Final Portfolio Value: {self.broker.getvalue():.2f}")
```

a. The next method is defined as:

```
def next(self):
    # check if we have enough data points
    self.day_counter += 1
    if self.day_counter < self.p.n_periods:
        return

    # check if the date is a Friday
    today = self.datas[0].datetime.date()
    if today.weekday() != 4:
        return
```

```

# find and print the current allocation
current_portf = {}
for data in self.datas:
    current_portf[data._name] = (
        self.positions[data].size * data.close[0]
    )

portf_df = pd.DataFrame(current_portf, index=[0])
print(f"Current allocation as of {today}")
print(portf_df / portf_df.sum(axis=1).squeeze())

# extract the past price data for each asset
price_dict = {}
for data in self.datas:
    price_dict[data._name] = (
        data.close.get(0, self.p.n_periods+1)
    )
prices_df = pd.DataFrame(price_dict)

# find the optimal portfolio weights
mu = mean_historical_return(prices_df)
S = CovarianceShrinkage(prices_df).ledoit_wolf()
ef = EfficientFrontier(mu, S)
weights = ef.max_sharpe(risk_free_rate=0)
print(f"Optimal allocation identified on {today}")
print(pd.DataFrame(ef.clean_weights(), index=[0]))

# create orders
for allocation in list(ef.clean_weights().items()):
    self.order_target_percent(data=allocation[0],
                             target=allocation[1])

```

3. Download the prices of the FAANG stocks and store the data feeds in a list:

```

TICKERS = ["META", "AMZN", "AAPL", "NFLX", "GOOG"]
data_list = []

for ticker in TICKERS:
    data = bt.feeds.YahooFinanceData(
        dataname=ticker,
        fromdate=datetime(2020, 1, 1),

```

```
        todate=datetime(2021, 12, 31)
    )
    data_list.append(data)
```

4. Set up the backtest:

```
cerebro = bt.Cerebro(stdstats=False)

cerebro.addstrategy(MeanVariancePortfStrategy)

for ind, ticker in enumerate(TICKERS):
    cerebro.adddata(data_list[ind], name=ticker)

cerebro.broker.setcash(1000.0)
cerebro.broker.addcommissioninfo(
    FractionalTradesCommission(commission=0)
)
cerebro.adddobserver(MyBuySell)
cerebro.adddobserver(bt.observers.Value)
```

5. Run the backtest:

```
backtest_result = cerebro.run()
```

Running the backtest generates the following log:

```
Initial Portfolio Value: 1000.00
Current allocation as of 2021-01-08
    META  AMZN  AAPL  NFLX  GOOG
0  NaN   NaN   NaN   NaN   NaN
Optimal allocation identified on 2021-01-08
    META      AMZN      AAPL  NFLX  GOOG
0  0.0   0.69394  0.30606   0.0   0.0
2021-01-11: Order Failed: AAPL
2021-01-11: BUY EXECUTED - Price: 157.40, Size: 4.36, Asset: AMZN, Cost:
686.40, Commission: 0.00
Current allocation as of 2021-01-15
    META  AMZN  AAPL  NFLX  GOOG
0  0.0   1.0   0.0   0.0   0.0
Optimal allocation identified on 2021-01-15
    META      AMZN      AAPL  NFLX  GOOG
0  0.0   0.81862  0.18138   0.0   0.0
2021-01-19: BUY EXECUTED - Price: 155.35, Size: 0.86, Asset: AMZN, Cost:
134.08, Commission: 0.00
```

```

2021-01-19: Order Failed: AAPL
Current allocation as of 2021-01-22
    META  AMZN  AAPL  NFLX  GOOG
0  0.0  1.0  0.0  0.0  0.0
Optimal allocation identified on 2021-01-22
    META      AMZN      AAPL  NFLX  GOOG
0  0.0  0.75501  0.24499  0.0  0.0
2021-01-25: SELL EXECUTED - Price: 166.43, Size: -0.46, Asset: AMZN, Cost:
71.68, Commission: 0.00
2021-01-25: Order Failed: AAPL
...
0  0.0  0.0  0.00943  0.0  0.99057
2021-12-20: Order Failed: GOOG
2021-12-20: SELL EXECUTED - Price: 167.82, Size: -0.68, Asset: AAPL, Cost:
110.92, Commission: 0.00
Final Portfolio Value: 1287.22

```

We will not spend time evaluating the strategy, as this would be very similar to what we did in the previous recipe. Thus, we leave it as a potential exercise for the reader. It could also be interesting to test the performance of this strategy against a benchmark $1/n$ portfolio.

It is worth mentioning that some of the orders failed. We will describe the reason for it in the following section.

How it works...

After importing the libraries, we defined the strategy using mean-variance optimization. In the `__init__` method, we defined a counter that we used to determine if we had enough data points to run the optimization routine. The selected 252 days is arbitrary and you can experiment with different values.

In the next method, there are multiple new components:

- We first add 1 to the day counter and check if we have enough observations. If not, we simply proceed to the next trading day.
- We extract the current date from the price data and check if it is a Friday. If not, we proceed to the next trading day.
- We calculate the current allocation by accessing the position size of each asset and multiplying it by the close price of the given day. Lastly, we divide each asset's worth by the total portfolio's value and print the weights.
- We need to extract the last 252 data points for each stock for our optimization routine. The `self.datas` object is an iterable containing all the data feeds we pass to Cerebro when setting up the backtest. We create a dictionary and populate it with arrays containing the 252 data points. We extract those using the `get` method. Then, we create a pandas DataFrame from the dictionary containing the prices.

- We find the weights maximizing the Sharpe ratio using the `pypfopt` library. Please refer to the previous chapter for more details. We also print the new weights.
- For each of the assets, we place a target order (using the `order_target_percent` method) with the target being the optimal portfolio weight. As we are working with multiple assets this time, we need to indicate for which asset we are placing an order. We do so by specifying the `data` argument.



Under the hood, `backtrader` uses the `array` module for storing the matrix-like objects.

In *Step 3*, we created a list containing all the data feeds. We simply iterated over the tickers of the FAANG stocks, downloaded the data for each one of them, and appended the object to the list.

In *Step 4*, we set up the backtest. A lot of the steps are already very familiar by now, including setting up the fractional shares commission scheme. The new component was adding the data, as we iteratively added each of the downloaded data feeds using the already covered `adddata` method. At this point, we also had to provide the name of the data feeds using the `name` argument.

In the very last step, we ran the backtest. As we have mentioned before, the new thing we can observe here is the failing orders. These are caused by the fact that we are calculating the portfolio weights on Friday using the close prices and preparing the orders on the same day. On Monday's market open, the prices are different, and not all the orders can be executed. We tried to account for that using fractional shares and setting the commission to 0, but the differences can still be too big for this simple approach to work. A possible solution would be to always keep some cash on the side to cover the potential price differences.

To do so, we could assume that we purchase the stocks with ~90% of our portfolio's worth while keeping the rest in cash. For that, we could use the `order_target_value` method. We could calculate the target value for each asset using the portfolio weights and 90% of the monetary value of our portfolio. Alternatively, we could use the `DiscreteAllocation` approach of `pypfopt`, which we mentioned in the previous chapter.

Summary

In this chapter, we have extensively covered the topic of backtesting. We started with the simpler approach, that is, vectorized backtesting. While it is not as rigorous and robust as the event-driven approach, it is often faster to implement and execute, due to its vectorized nature. Afterward, we combined the exploration of the event-driven backtesting framework with the knowledge we obtained in the previous chapters, for example, calculating various technical indicators and finding the optimal portfolio weights.

We spent the most time using the backtrader library, due to its popularity and flexibility when it comes to implementing various scenarios. However, there are many alternative backtesting libraries on the market. You might also want to investigate the following:

- `vectorbt` (<https://github.com/polakowo/vectorbt>): A pandas-based library for efficient backtesting of trading strategies at scale. The author of the library also offers a pro (paid) version of the library with more features and improved performance.
- `bt` (<https://github.com/pmorissette/bt>): A library offering a framework based on reusable and flexible blocks containing the strategy's logic. It supports multiple instruments and outputs detailed statistics and charts.
- `backtesting.py` (<https://github.com/kernc/backtesting.py>): A backtesting framework built on top of backtrader.
- `fastquant` (<https://github.com/enzoampil/fastquant>): A wrapper library around backtrader that aims to reduce the amount of boilerplate code we need to write in order to run a backtest for popular trading strategies, for example, the moving average crossover.
- `zipline` (<https://github.com/quantopian/zipline> / <https://github.com/stefan-jansen/zipline-reloaded>): The library used to be the most popular (based on GitHub stars) and probably the most complex of the open-source backtesting libraries. However, as we have already mentioned, Quantopian was closed and the library is not maintained anymore. You can use the fork (`zipline-reloaded`) maintained by Stefan Jansen.

Backtesting is a fascinating field and there is much more to learn about it. Below, you can also find some very interesting references for more robust approaches to backtesting:

- Bailey, D. H., Borwein, J., Lopez de Prado, M., & Zhu, Q. J. (2016). “The probability of backtest overfitting.” *Journal of Computational Finance, forthcoming*.
- Bailey, D. H., & De Prado, M. L. (2014). “The deflated Sharpe ratio: correcting for selection bias, backtest overfitting, and non-normality.” *The Journal of Portfolio Management, 40* (5), 94-107.
- Bailey, D. H., Borwein, J., Lopez de Prado, M., & Zhu, Q. J. (2014). “Pseudo-mathematics and financial charlatanism: The effects of backtest overfitting on out-of-sample performance.” *Notices of the American Mathematical Society, 61* (5), 458-471.
- De Prado, M. L. (2018). *Advances in Financial Machine Learning*. John Wiley & Sons.

13

Applied Machine Learning: Identifying Credit Default

In recent years, we have witnessed machine learning gaining more and more popularity in solving traditional business problems. Every so often, a new algorithm is published, beating the current state of the art. It is only natural for businesses (in all industries) to try to leverage the incredible powers of machine learning in their core functionalities.

Before specifying the task we will be focusing on in this chapter, we provide a brief introduction to the field of machine learning. The machine learning domain can be broken down into two main areas: supervised learning and unsupervised learning. In the former, we have a target variable (label), which we try to predict as accurately as possible. In the latter, there is no target, and we try to use different techniques to draw some insights from the data.

We can further break down supervised problems into regression problems (where a target variable is a continuous number, such as income or the price of a house) and classification problems (where the target is a class, either binary or multi-class). An example of unsupervised learning is clustering, which is often used for customer segmentation.

In this chapter, we tackle a binary classification problem set in the financial industry. We work with a dataset contributed to the UCI Machine Learning Repository, which is a very popular data repository. The dataset used in this chapter was collected in a Taiwanese bank in October 2005. The study was motivated by the fact that—at that time—more and more banks were giving credit (either cash or via credit cards) to willing customers. On top of that, more people, regardless of their repayment capabilities, accumulated significant amounts of debt. All of this led to situations in which some people were unable to repay their outstanding debts. In other words, they defaulted on their loans.

The goal of the study was to use some basic information about customers (such as gender, age, and education level), together with their past repayment history, to predict which of them were likely to default. The setting can be described as follows—using the previous 6 months of repayment history (April–September 2005), we try to predict whether the customer will default in October 2005. Naturally, such a study could be generalized to predict whether a customer will default in the next month, within the next quarter, and so on.

By the end of this chapter, you will be familiar with a real-life approach to a machine learning task, from gathering and cleaning data to building and tuning a classifier. Another takeaway is understanding the general approach to machine learning projects, which can then be applied to many different tasks, be it churn prediction or estimating the price of new real estate in a neighborhood.

In this chapter, we focus on the following recipes:

- Loading data and managing data types
- Exploratory data analysis
- Splitting data into training and test sets
- Identifying and dealing with missing values
- Encoding categorical variables
- Fitting a decision tree classifier
- Organizing the project with pipelines
- Tuning hyperparameters using grid search and cross-validation

Loading data and managing data types

In this recipe, we show how to load a dataset from a CSV file into Python. The very same principles can be used for other file formats as well, as long as they are supported by pandas. Some popular formats include Parquet, JSON, XLM, Excel, and Feather.



pandas has a very consistent API, which makes finding its functions much easier. For example, all functions used for loading data from various sources have the syntax `pd.read_xxx`, where `xxx` should be replaced by the file format.

We also show how certain data type conversions can significantly reduce the size of DataFrames in the memory of our computers. This can be especially important when working with large datasets (GBs or TBs), which can simply not fit into memory unless we optimize their usage.

In order to present a more realistic scenario (including messy data, missing values, and so on) we applied some transformations to the original dataset. For more information on those changes, please refer to the accompanying GitHub repository.

How to do it...

Execute the following steps to load a dataset from a CSV file into Python:

1. Import the libraries:

```
import pandas as pd
```

2. Load the data from the CSV file:

```
df = pd.read_csv("../Datasets/credit_card_default.csv",
                 na_values="")
df
```

Running the snippet generates the following preview of the dataset:

	limit_bal	sex	education	marriage	age	payment_status_sep	payment_status_aug
0	20000	Female	University	Married	24.0	Payment delayed 2 months	Payment delayed 2 months
1	120000	Female	University	Single	26.0	Payed duly	Payment delayed 2 months
2	90000	Female	University	Single	34.0	Unknown	Unknown
3	50000	Female	University	Married	37.0	Unknown	Unknown
4	50000	Male	University	Married	57.0	Payed duly	Unknown
...
29995	220000	NaN	High school	Married	39.0	Unknown	Unknown
29996	150000	Male	High school	Single	43.0	Payed duly	Payed duly
29997	30000	Male	University	Single	37.0	Payment delayed 4 months	Payment delayed 3 months
29998	80000	Male	High school	Married	41.0	Payment delayed 1 month	Payed duly
29999	50000	Male	University	Married	46.0	Unknown	Unknown

Figure 13.1: Preview of the dataset. Not all columns were displayed

The DataFrame has 30,000 rows and 24 columns. It contains a mix of numeric and categorical variables.

3. View the summary of the DataFrame:

```
df.info()
```

Running the snippet generates the following summary:

```
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 24 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   limit_bal        30000 non-null   int64  
 1   sex              29850 non-null   object  
 2   education        29850 non-null   object  
 3   marriage         29850 non-null   object  
 4   age               29850 non-null   float64 
 5   payment_status_sep 30000 non-null   object  
 6   payment_status_aug 30000 non-null   object  
 7   payment_status_jul 30000 non-null   object  
 8   payment_status_jun 30000 non-null   object  
 9   payment_status_may 30000 non-null   object  
 10  payment_status_apr 30000 non-null   object  
 11  bill_statement_sep 30000 non-null   int64  
 12  bill_statement_aug 30000 non-null   int64  
 13  bill_statement_jul 30000 non-null   int64  
 14  bill_statement_jun 30000 non-null   int64  
 15  bill_statement_may 30000 non-null   int64  
 16  bill_statement_apr 30000 non-null   int64  
 17  previous_payment_sep 30000 non-null   int64  
 18  previous_payment_aug 30000 non-null   int64  
 19  previous_payment_jul 30000 non-null   int64  
 20  previous_payment_jun 30000 non-null   int64  
 21  previous_payment_may 30000 non-null   int64  
 22  previous_payment_apr 30000 non-null   int64  
 23  default_payment_next_month 30000 non-null   int64  
dtypes: float64(1), int64(14), object(9)
memory usage: 5.5+ MB
```

In the summary, we can see information about the columns and their data types, the number of non-null (in other words, non-missing) values, the memory usage, and so on.

We can also observe a few distinct data types: floats (floating-point numbers, such as 3.42), integers, and objects. The last ones are the pandas representation of string variables. The number next to `float` and `int` indicates how many bits this type uses to represent a particular value. The default types use 64 bits (or 8 bytes) of memory.



The basic `int8` type covers integers in the following range: -128 to 127. `uint8` stands for unsigned integer and covers the same total span, but only the non-negative values, that is, 0 to 255. By knowing the range of values covered by specific data types (please refer to the link in the *See also* section), we can try to optimize allocated memory. For example, for features such as the month of purchase (represented by numbers in the range 1-12), there is no point in using the default `int64`, as a much smaller type would suffice.

4. Define a function for inspecting the exact memory usage of a DataFrame:

```
def get_df_memory_usage(df, top_columns=5):
    print("Memory usage ----")
    memory_per_column = df.memory_usage(deep=True) / (1024 ** 2)
    print(f"Top {top_columns} columns by memory (MB):")
    print(memory_per_column.sort_values(ascending=False) \
          .head(top_columns))
    print(f"Total size: {memory_per_column.sum():.2f} MB")
```

We can now apply the function to our DataFrame:

```
get_df_memory_usage(df, 5)
```

Running the snippet generates the following output:

```
Memory usage ----
Top 5 columns by memory (MB):
education           1.965001
payment_status_sep  1.954342
payment_status_aug  1.920288
payment_status_jul  1.916343
payment_status_jun  1.904229
dtype: float64
Total size: 20.47 MB
```

In the output, we can see that the 5.5+ MB reported by the `info` method turned out to be almost 4 times more. This is still very small in terms of current machines' capabilities, however, the memory-saving principles we show in this chapter apply just as well to DataFrames measured in gigabytes.

5. Convert the columns with the object data type into the category type:

```
object_columns = df.select_dtypes(include="object").columns
df[object_columns] = df[object_columns].astype("category")

get_df_memory_usage(df)
```

Running the snippet generates the following overview:

```
Memory usage ----
Top 5 columns by memory (MB):
bill_statement_sep      0.228882
bill_statement_aug      0.228882
previous_payment_apr    0.228882
previous_payment_may    0.228882
previous_payment_jun    0.228882
dtype: float64
Total size: 3.70 MB
```

Just by converting the object columns into a pandas-native categorical representation, we managed to reduce the size of the DataFrame by ~80%!

6. Downcast the numeric columns to integers:

```
numeric_columns = df.select_dtypes(include="number").columns
for col in numeric_columns:
    df[col] = pd.to_numeric(df[col], downcast="integer")

get_df_memory_usage(df)
```

Running the snippet generates the following overview:

```
Memory usage ----
Top 5 columns by memory (MB):
age                      0.228882
bill_statement_sep        0.114441
limit_bal                 0.114441
previous_payment_jun     0.114441
previous_payment_jul     0.114441
dtype: float64
Total size: 2.01 MB
```

In the summary, we can see that after a few data type conversions, the column that takes up the most memory is the one containing customers' ages (you can see that in the output of `df.info()`, not shown here for brevity). That is because it is encoded using a `float` data type and downcasting using the `integer` setting was not applied to `float` columns.

7. Downcast the age column using the float data type:

```
df[ "age" ] = pd.to_numeric(df[ "age" ], downcast="float")
get_df_memory_usage(df)
```

Running the snippet generates the following overview:

```
Memory usage ----
Top 5 columns by memory (MB):
bill_statement_sep      0.114441
limit_bal               0.114441
previous_payment_jun    0.114441
previous_payment_jul    0.114441
previous_payment_aug    0.114441
dtype: float64
Total size: 1.90 MB
```

Using various data type conversions, we have managed to reduce the memory size of our DataFrame from 20.5 MB to 1.9 MB, which is a 91% reduction.

How it works...

After importing pandas, we loaded the CSV file by using the `pd.read_csv` function. When doing so, we indicated that empty strings should be interpreted as missing values.

In *Step 3*, we displayed a summary of the DataFrame to inspect its contents. To get a better understanding of the dataset, we provide a simplified description of the variables:

- `limit_bal`—the amount of the given credit (NT dollars)
- `sex`—biological sex
- `education`—level of education
- `marriage`—marital status
- `age`—age of the customer
- `payment_status_{month}`—status of payments in one of the previous 6 months
- `bill_statement_{month}`—the number of bill statements (NT dollars) in one of the previous 6 months
- `previous_payment_{month}`—the number of previous payments (NT dollars) in one of the previous 6 months
- `default_payment_next_month`—the target variable indicating whether the customer defaulted on the payment in the following month

In general, pandas tries to load and store data as efficiently as possible. It automatically assigns data types (which we can inspect by using the `dtypes` method of a pandas DataFrame). However, there are some tricks that can lead to much better memory allocation, which definitely makes working with larger tables (in hundreds of MBs, or even GBs) easier and more efficient.

In *Step 4*, we defined a function for inspecting the exact memory usage of a DataFrame. The `memory_usage` method returns a pandas Series with the memory usage (in bytes) for each of the DataFrame's columns. We converted the output into MBs to make it easier to understand.



When using the `memory_usage` method, we specified `deep=True`. That is because the `object` data type, unlike other dtypes (short for data types), does not have a fixed memory allocation for each cell. In other words, as the `object` dtype usually corresponds to text, it means that the amount of memory used depends on the number of characters in each cell. Intuitively, the more characters in a string, the more memory that cell uses.

In *Step 5*, we leveraged a special data type called `category` to reduce the DataFrame's memory usage. The underlying idea is that string variables are encoded as integers, and pandas uses a special mapping dictionary to decode them back into their original form. This is especially useful when dealing with a limited number of distinct values, for example, certain levels of education, country of origin, and so on. To save memory, we first identified all the columns with the `object` data type using the `select_dtypes` method. Then, we changed the data type of those columns from `object` to `category`. We did so using the `astype` method.



We should know when it is actually profitable (from the memory's perspective) to use the `category` data type. A rule of thumb is to use it for variables with a ratio of unique observations to the overall number of observations lower than 50%.

In *Step 6*, we used the `select_dtypes` method to identify all numeric columns. Then, using a `for` loop iterating over the identified columns, we converted the values to numeric using the `pd.to_numeric` function. This might strike as odd, given that we first identified the numeric columns and then converted them to numeric again. However, the crucial part is the `downcast` argument of the function. By passing the "integer" value, we have optimized the memory usage of all the integer columns by downcasting the default `int64` data type to smaller alternatives (`int32` and `int8`).

Even though we applied the function to all numeric columns, only the ones that contained integers were successfully transformed. That is why in *Step 7* we additionally downcasted the `float` column containing the clients' ages.

There's more...

In this recipe, we have mentioned how to optimize the memory usage of a pandas DataFrame. We first loaded the data into Python, then we inspected the columns, and at the end we converted the data types of some columns to reduce memory usage. However, such an approach might not be possible, as the data might simply not fit into memory in the first place.

If that is the case, we can also try the following:

- Read the dataset in chunks (by using the `chunk` argument of `pd.read_csv`). For example, we could load just the first 100 rows of data.

- Read only the columns we actually need (by using the `usecols` argument of `pd.read_csv`).
- While loading the data, use the `column_dtypes` argument to define the data types used for each of the columns.

To illustrate, we can use the following snippet to load our dataset and while doing so indicate that the selected three columns should have a `category` data type:

```
column_dtypes = {
    "education": "category",
    "marriage": "category",
    "sex": "category"
}
df_cat = pd.read_csv("../Datasets/credit_card_default.csv",
                     na_values="", dtype=column_dtypes)
```

If all of those approaches fail, we should not give up. While `pandas` is definitely the gold standard of working with tabular data in Python, we can leverage the power of some alternative libraries, which were built specifically for such a case. Below you can find a list of libraries you could use when working with large volumes of data:

- `Dask`: an open-source library for distributed computing. It facilitates running many computations at the same time, either on a single machine or on clusters of CPUs. Under the hood, the library breaks down a single large data processing job into many smaller tasks, which are then handled by `numpy` or `pandas`. As the last step, the library reassembles the results into a coherent whole.
- `Modin`: a library designed to parallelize `pandas` DataFrames by automatically distributing the computation across all of the system's available CPU cores. The library divides an existing DataFrame into different parts such that each part can be sent to a different CPU core.
- `Vaex`: an open-source DataFrame library specializing in lazy out-of-core DataFrames. Vaex requires negligible amounts of RAM for inspecting and interacting with a dataset of arbitrary size, all thanks to combining the concepts of lazy evaluations and memory mapping.
- `datatable`: an open-source library for manipulating 2-dimensional tabular data. In many ways, it is similar to `pandas`, with special emphasis on speed and the volume of data (up to 100 GB) while using a single-node machine. If you have worked with R, you might already be familiar with the related package called `data.table`, which is R users' go-to package when it comes to the fast aggregation of large data.
- `cuDF`: a GPU DataFrame library that is part of NVIDIA's RAPIDS, a data science ecosystem spanning multiple open-source libraries and leveraging the power of GPUs. `cuDF` allows us to use a `pandas`-like API to benefit from the performance boost without going into the details of CUDA programming.
- `polars`: an open-source DataFrame library that achieves phenomenal computation speed by leveraging Rust (programming language) with Apache Arrow as its memory model.

See also

Additional resources:

- Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.
- Yeh, I. C. & Lien, C. H. (2009). “The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients.” *Expert Systems with Applications*, 36(2), 2473-2480. <https://doi.org/10.1016/j.eswa.2007.12.020>.
- List of different data types used in Python: <https://numpy.org/doc/stable/user/basics.types.html#>.

Exploratory data analysis

The second step of a data science project is to carry out **Exploratory Data Analysis (EDA)**. By doing so, we get to know the data we are supposed to work with. This is also the step during which we test the extent of our domain knowledge. For example, the company we are working for might assume that the majority of its customers are people between the ages of 18 and 25. But is this actually the case? While doing EDA we might also run into some patterns that we do not understand, which are then a starting point for a discussion with our stakeholders.

While doing EDA, we can try to answer the following questions:

- What kind of data do we actually have, and how should we treat different data types?
- What is the distribution of the variables?
- Are there outliers in the data and how can we treat them?
- Are any transformations required? For example, some models work better with (or require) normally distributed variables, so we might want to use techniques such as log transformation.
- Does the distribution vary per group (for example, sex or education level)?
- Do we have cases of missing data? How frequent are these, and in which variables do they occur?
- Is there a linear relationship (correlation) between some variables?
- Can we create new features using the existing set of variables? An example might be deriving an hour/minute from a timestamp, a day of the week from a date, and so on.
- Are there any variables that we can remove as they are not relevant for the analysis? An example might be a randomly generated customer identifier.

Naturally, this list is non-exhaustive and carrying out the analysis might spark more questions than we initially had. EDA is extremely important in all data science projects, as it enables analysts to develop an understanding of the data, facilitates asking better questions, and makes it easier to pick modeling approaches suitable for the type of data being dealt with.

In real-life cases, it makes sense to first carry out a univariate analysis (one feature at a time) for all relevant features to get a good understanding of them. Then, we can proceed to multivariate analysis, that is, comparing distributions per group, correlations, and so on. For brevity, we only show selected analysis approaches to selected features, but a deeper analysis is highly encouraged.

Getting ready

We continue with exploring the data we loaded in the previous recipe.

How to do it...

Execute the following steps to carry out the EDA of the loan default dataset:

1. Import the libraries:

```
import pandas as pd
import numpy as np
import seaborn as sns
```

2. Get summary statistics of the numeric variables:

```
df.describe().transpose().round(2)
```

Running the snippet generates the following summary table:

	count	mean	std	min	25%	50%	75%	max
limit_bal	30000.0	167484.32	129747.66	10000.0	50000.00	140000.0	240000.00	1000000.0
age	29850.0	35.49	9.22	21.0	28.00	34.0	41.00	79.0
bill_statement_sep	30000.0	51223.33	73635.86	-165580.0	3558.75	22381.5	67091.00	964511.0
bill_statement_aug	30000.0	49179.08	71173.77	-69777.0	2984.75	21200.0	64006.25	983931.0
bill_statement_jul	30000.0	47013.15	69349.39	-157264.0	2666.25	20088.5	60164.75	1664089.0
bill_statement_jun	30000.0	43262.95	64332.86	-170000.0	2326.75	19052.0	54506.00	891586.0
bill_statement_may	30000.0	40311.40	60797.16	-81334.0	1763.00	18104.5	50190.50	927171.0
bill_statement_apr	30000.0	38871.76	59554.11	-339603.0	1256.00	17071.0	49198.25	961664.0
previous_payment_sep	30000.0	5663.58	16563.28	0.0	1000.00	2100.0	5006.00	873552.0
previous_payment_aug	30000.0	5921.16	23040.87	0.0	833.00	2009.0	5000.00	1684259.0
previous_payment_jul	30000.0	5225.68	17606.96	0.0	390.00	1800.0	4505.00	896040.0
previous_payment_jun	30000.0	4826.08	15666.16	0.0	296.00	1500.0	4013.25	621000.0
previous_payment_may	30000.0	4799.39	15278.31	0.0	252.50	1500.0	4031.50	426529.0
previous_payment_apr	30000.0	5215.50	17777.47	0.0	117.75	1500.0	4000.00	528666.0
default_payment_next_month	30000.0	0.22	0.42	0.0	0.00	0.0	0.00	1.0

Figure 13.2: Summary statistics of the numeric variables

3. Get summary statistics of the categorical variables:

```
df.describe(include="object").transpose()
```

Running the snippet generates the following summary table:

	count	unique	top	freq
sex	29850	2	Female	18027
education	29850	4	University	13960
marriage	29850	3	Single	15891
payment_status_sep	30000	10	Unknown	17496
payment_status_aug	30000	10	Unknown	19512
payment_status_jul	30000	10	Unknown	19849
payment_status_jun	30000	10	Unknown	20803
payment_status_may	30000	9	Unknown	21493
payment_status_apr	30000	9	Unknown	21181

Figure 13.3: Summary statistics of the categorical variables

4. Plot the distribution of age and split it by sex:

```
ax = sns.kdeplot(data=df, x="age",
                  hue="sex", common_norm=False,
                  fill=True)
ax.set_title("Distribution of age")
```

Running the snippet generates the following plot:

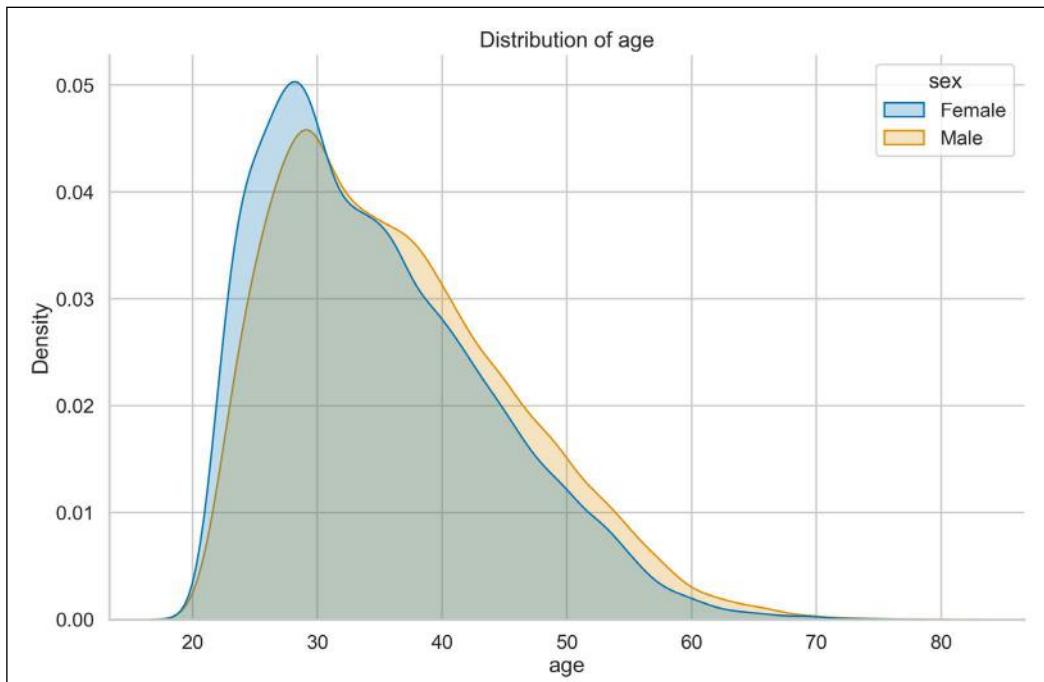


Figure 13.4: The KDE plot of age, grouped by sex

By analyzing the **kernel density estimate (KDE)** plot, we can say there is not much difference in the shape of the distribution per sex. The female sample is slightly younger, on average.

5. Create a pairplot of selected variables:

```
COLS_TO_PLOT = ["age", "limit_bal", "previous_payment_sep"]

pair_plot = sns.pairplot(df[COLS_TO_PLOT], kind="reg",
                         diag_kind="kde", height=4,
                         plot_kws={"line_kws":{"color":"red"}})
pair_plot.fig.suptitle("Pairplot of selected variables")
```

Running the snippet generates the following plot:

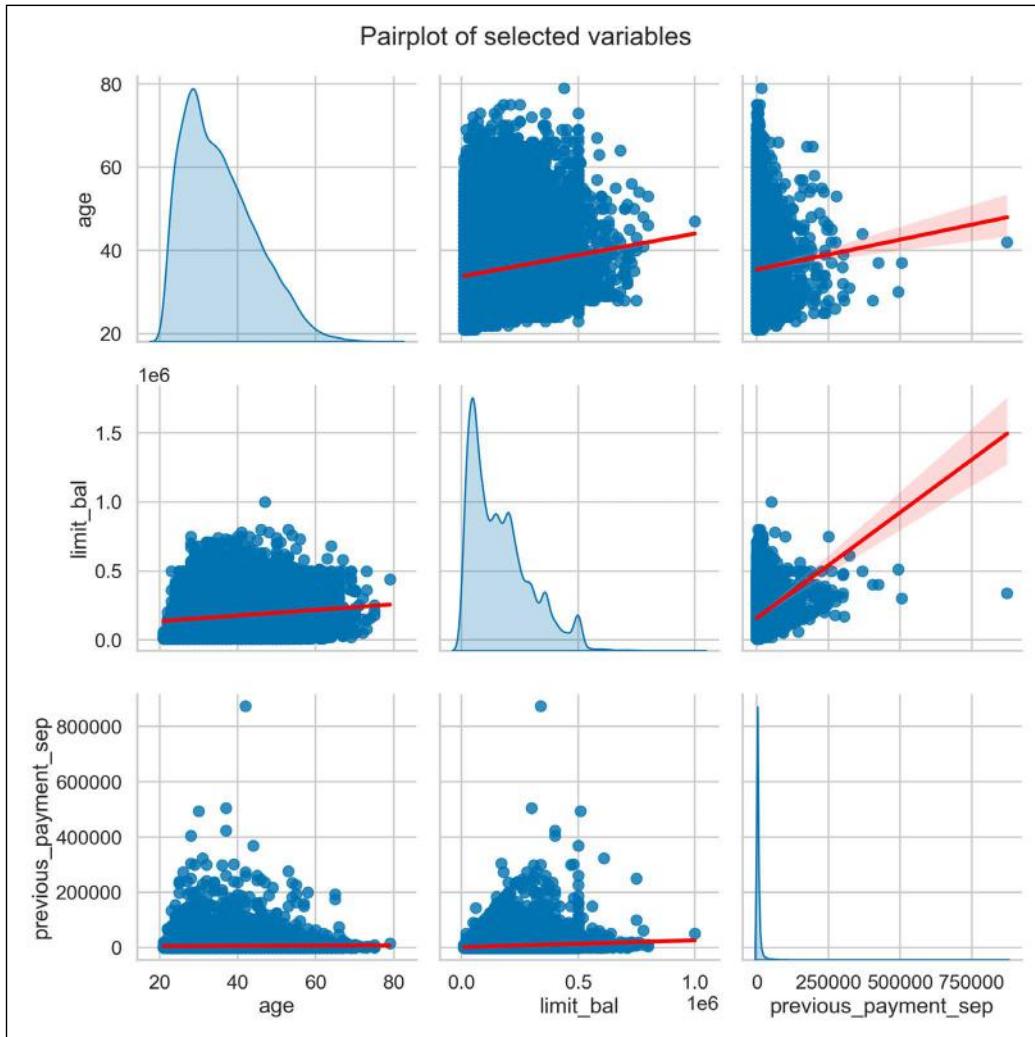


Figure 13.5: A pairplot with KDE plots on the diagonal and fitted regression lines in each scatterplot

We can make a few observations from the created pairplot:

- The distribution of `previous_payment_sep` is highly skewed—it has a very long tail.
- Connected to the previous point, we can observe some very extreme values of `previous_payment_sep` in the scatterplots.
- It is difficult to draw conclusions from the scatterplots, as there are 30,000 observations on each of them. When plotting such volumes of data, we could use transparent markers to better visualize the density of the observation in certain areas.
- The outliers can have a significant impact on the regression lines.

Additionally, we can separate the sexes by specifying the hue argument:

```
pair_plot = sns.pairplot(data=df,
                         x_vars=COLS_TO_PLOT,
                         y_vars=COLS_TO_PLOT,
                         hue="sex",
                         height=4)
pair_plot.fig.suptitle("Pairplot of selected variables")
```

Running the snippet generates the following plot:

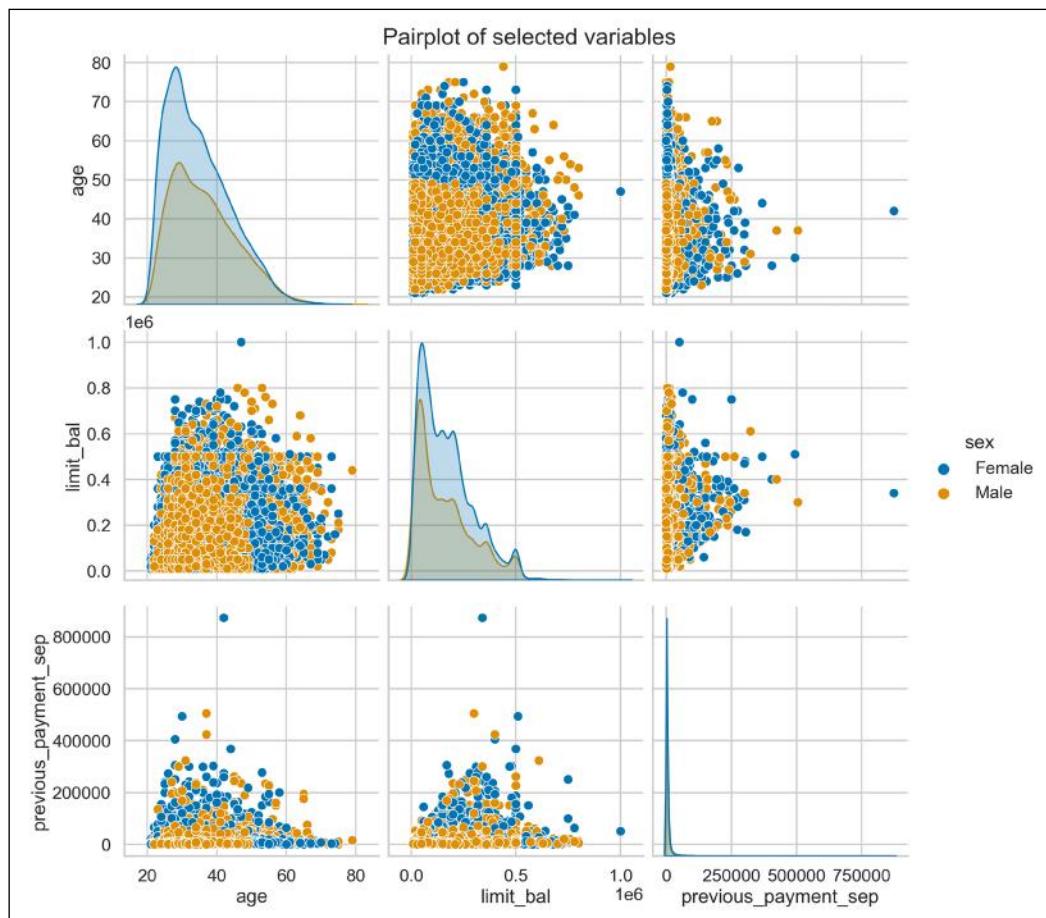


Figure 13.6: The pairplot with separate markers for each sex

While we can gain some more insights from the diagonal plots with the split per sex, the scatterplots are still quite unreadable due to the sheer volume of plotted data.

As a potential solution, we could randomly sample from the entire dataset and only plot the selected observations. A possible downside of that approach is that we might miss some observations with extreme values (outliers).

6. Analyze the relationship between age and limit balance:

```
ax = sns.jointplot(data=df, x="age", y="limit_bal",
                    hue="sex", height=10)
ax.fig.suptitle("Age vs. limit balance")
```

Running the snippet generates the following plot:

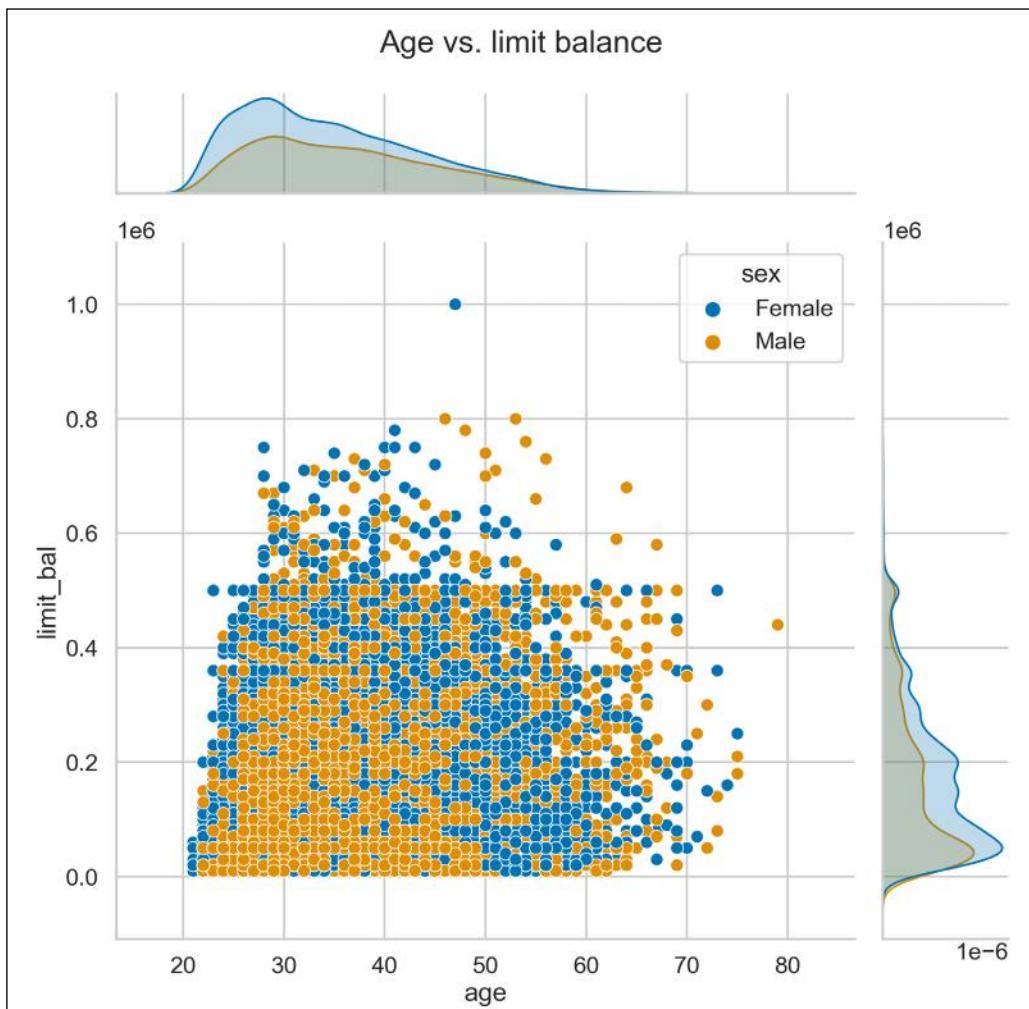


Figure 13.7: A joint plot showing the relationship between age and limit balance, grouped by sex

A joint plot contains quite a lot of useful information. First of all, we can see the relationship between two variables on the scatterplot. Then, we can also investigate the distributions of the two variables individually using the KDE plots along the axes (we can also plot histograms instead).

7. Define and run a function for plotting the correlation heatmap:

```
def plot_correlation_matrix(corr_mat):  
    sns.set(style="white")  
    mask = np.zeros_like(corr_mat, dtype=bool)  
    mask[np.triu_indices_from(mask)] = True  
    fig, ax = plt.subplots()  
    cmap = sns.diverging_palette(240, 10, n=9, as_cmap=True)  
    sns.heatmap(corr_mat, mask=mask, cmap=cmap,  
                vmax=.3, center=0, square=True,  
                linewidths=.5, cbar_kws={"shrink": .5},  
                ax=ax)  
    ax.set_title("Correlation Matrix", fontsize=16)  
    sns.set(style="darkgrid")  
  
corr_mat = df.select_dtypes(include="number").corr()  
plot_correlation_matrix(corr_mat)
```

Running the snippet generates the following plot:

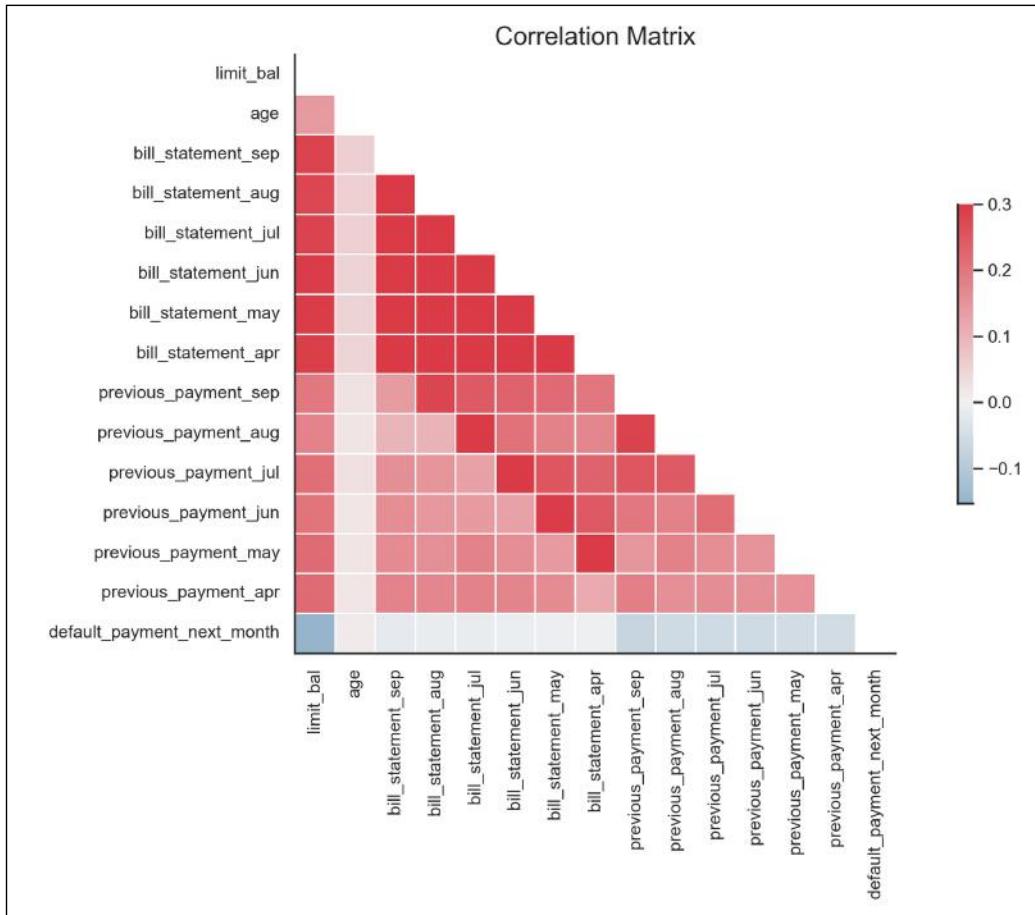


Figure 13.8: Correlation heatmap of the numeric features

We can see that age seems to be uncorrelated to any of the other features.

8. Analyze the distribution of age in groups using box plots:

```
ax = sns.boxplot(data=df, y="age", x="marriage", hue="sex")
ax.set_title("Distribution of age")
```

Running the snippet generates the following plot:

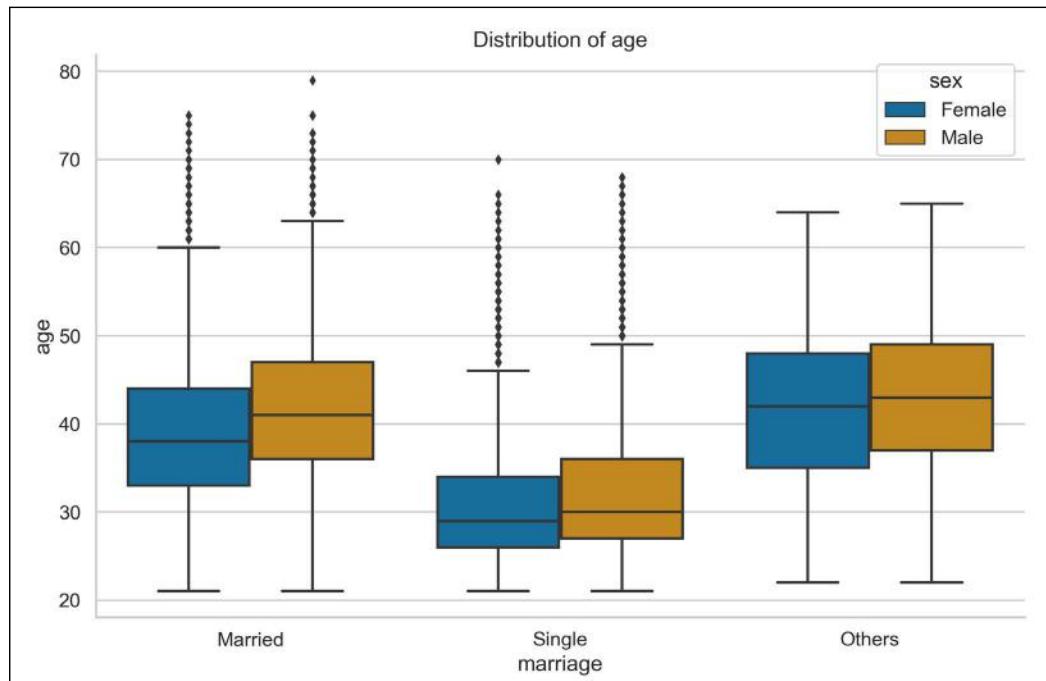


Figure 13.9: Distribution of age by marital status and sex

The distributions seem quite similar within marital groups, with men always having a higher median age.

9. Plot the distribution of limit balance for each sex and education level:

```
ax = sns.violinplot(x="education", y="limit_bal",
                     hue="sex", split=True, data=df)
ax.set_title(
    "Distribution of limit balance per education level",
    fontsize=16
)
```

Running the snippet generates the following plot:

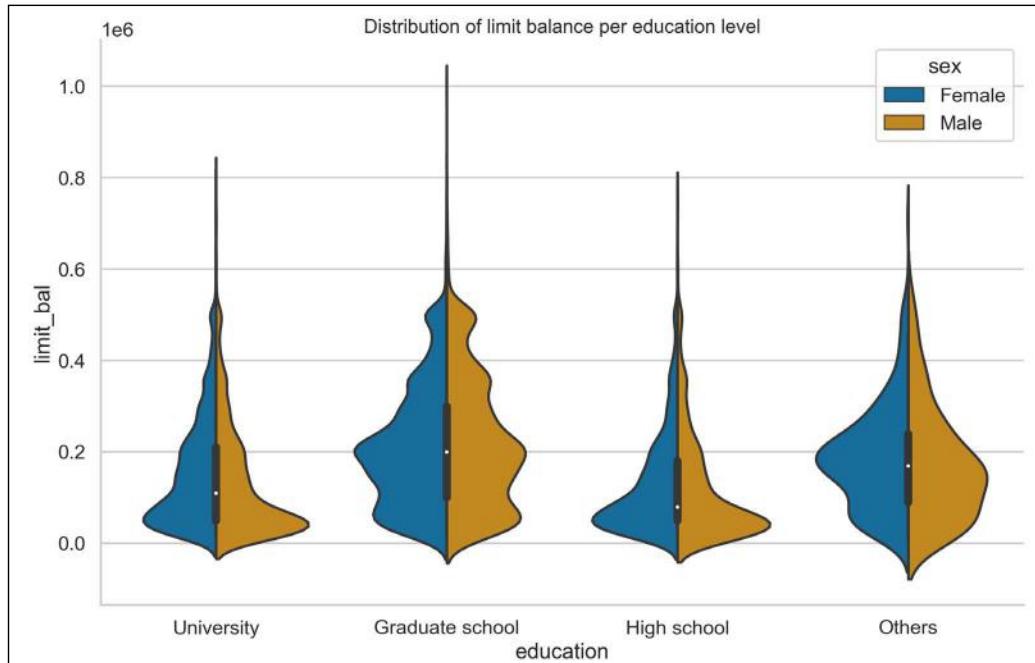


Figure 13.10: Distribution of limit balance by education level and sex

Inspecting the plot reveals a few interesting patterns:

- The largest balance appears in the group with the *Graduate school* level of education.
- The shape of the distribution is different per education level: the *Graduate school* level resembles the *Others* category, while the *High school* level is similar to the *University* level.
- In general, there are few differences between the sexes.

10. Investigate the distribution of the target variable per sex and education level:

```
ax = sns.countplot("default_payment_next_month", hue="sex",
                   data=df, orient="h")
ax.set_title("Distribution of the target variable", fontsize=16)
```

Running the snippet generates the following plot:

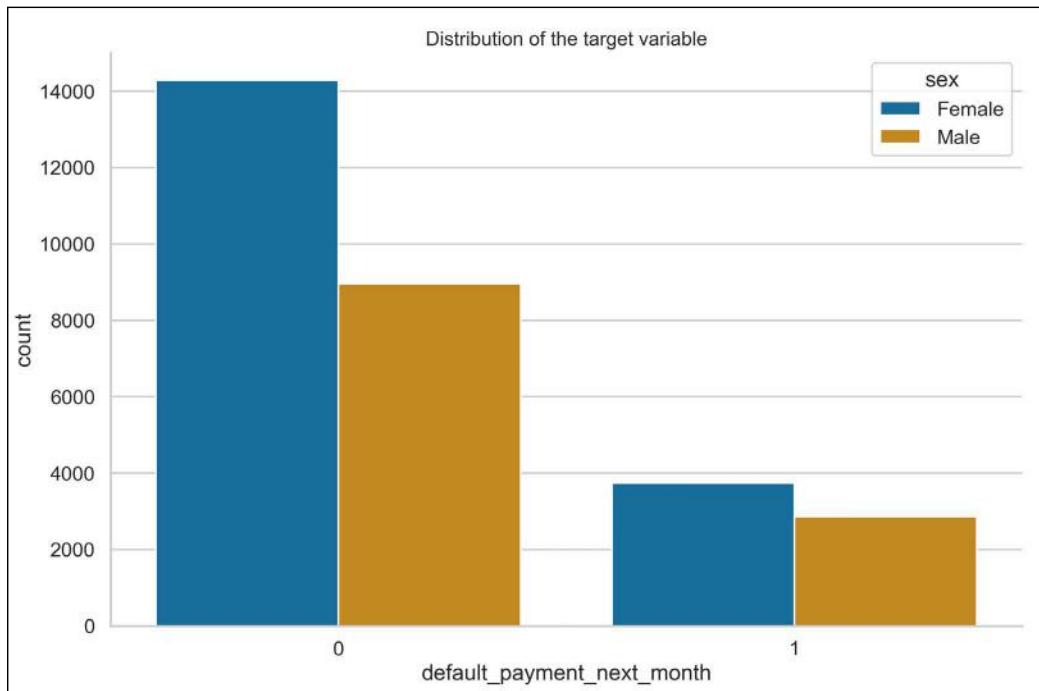


Figure 13.11: Distribution of the target variable by sex

By analyzing the plot, we can say that the percentage of defaults is higher among male customers.

11. Investigate the percentage of defaults per education level:

```
ax = df.groupby("education")["default_payment_next_month"] \
    .value_counts(normalize=True) \
    .unstack() \
    .plot(kind="barh", stacked=True)
ax.set_title("Percentage of default per education level",
            fontsize=16)
ax.legend(title="Default", bbox_to_anchor=(1,1))
```

Running the snippet generates the following plot:

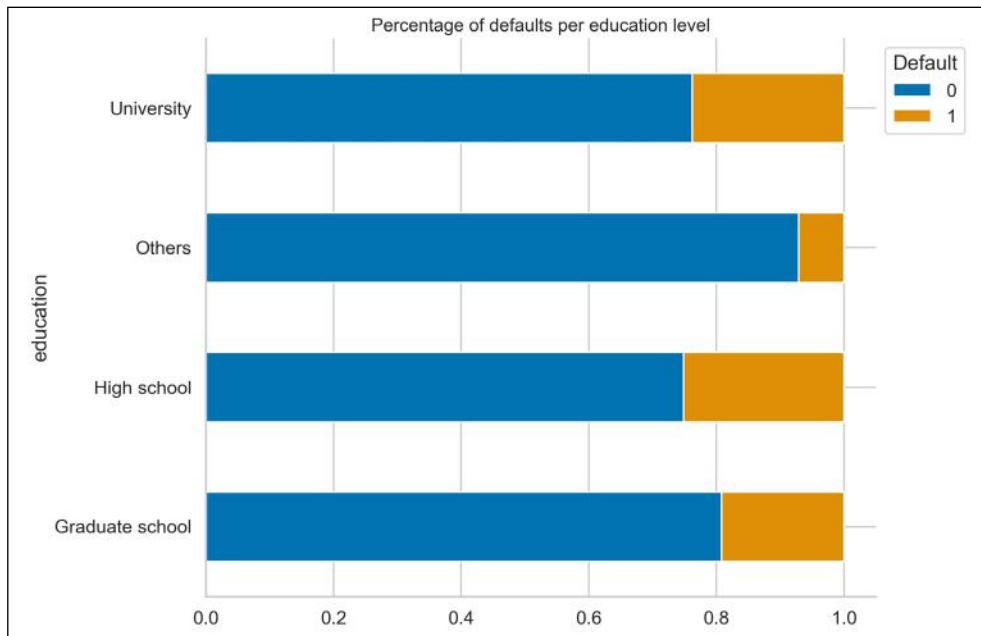


Figure 13.12: Percentage of defaults by education level

Relatively speaking, most defaults happen among customers with a high-school education, while the fewest defaults happen in the *Others* category.

How it works...

In the previous recipe, we already explored two DataFrame methods that are useful for starting exploratory data analysis: `shape` and `info`. We can use them to quickly learn the shape of the dataset (number of rows and columns), what data types are used for representing each feature, and so on.



In this recipe, we are mostly using the `seaborn` library, as it is the go-to library when it comes to exploring data. However, we could use alternative plotting libraries. The `plot` method of a pandas DataFrame is quite powerful and allows for quickly visualizing our data. Alternatively, we could use `plotly` (and its `plotly.express` module) to create fully interactive data visualizations.

In this recipe, we started the analysis by using a very simple yet powerful method of a pandas DataFrame—`describe`. It printed summary statistics, such as the count, mean, min/max, and quartiles of all the numeric variables in the DataFrame. By inspecting these metrics, we could infer the value range of a certain feature, or whether the distribution is skewed (by looking at the difference between the mean and median). Also, we could easily spot values outside the plausible range, for example, a negative or very young/old age.



We can include additional percentiles in the `describe` method by passing an extra argument, for example, `percentiles=[.99]`. In this case, we added the 99th percentile.

The count metric represents the number of non-null observations, so it is also a way to determine which numeric features contain missing values. Another way of investigating the presence of missing values is by running `df.isnull().sum()`. For more information on missing values, please see the *Identifying and dealing with missing values* recipe.

In Step 3, we added the `include="object"` argument while calling the `describe` method to inspect the categorical features separately. The output was different from the numeric features: we could see the count, the number of unique categories, which one was the most frequent, and how many times it appeared in the dataset.



We can use `include="all"` to display the summary metrics for all features—only the metrics available for a given data type will be present, while the rest will be filled with `NA` values.

In Step 4, we showed a way of investigating the distribution of a variable, in this case, the age of the customers. To do so, we created a KDE plot. It is a method of visualizing the distribution of a variable, very similar to a traditional histogram. KDE represents the data using a continuous probability density curve in one or more dimensions. One of its advantages over a histogram is that the resulting plot is less cluttered and easier to interpret, especially when considering multiple distributions at once.



A common source of confusion around the KDE plots is about the units on the density axis. In general, the kernel density estimation results in a probability distribution. However, the height of the curve at each point gives a density, instead of the probability. We can obtain a probability by integrating the density across a certain range. The KDE curve is normalized so that the integral over all possible values is equal to 1. This means that the scale of the density axis depends on the data values. To take it a step further, we can decide how to normalize the density when we are dealing with multiple categories in one plot. If we use `common_norm=True`, each density is scaled by the number of observations so that the total area under all curves sums to 1. Otherwise, the density of each category is normalized independently.

Together with a histogram, the KDE plot is one of the most popular methods of inspecting the distribution of a single feature. To create a histogram, we can use the `sns.histplot` function. Alternatively, we can use the `plot` method of a `pandas` DataFrame, while specifying `kind="hist"`. We show examples of creating histograms in the accompanying Jupyter notebook (available on GitHub).

An extension of this analysis can be done by using a pairplot. It creates a matrix of plots, where the diagonal shows the univariate histograms or KDE plots, while the off-diagonal plots are scatterplots of two features. This way, we can also try to see if there is a relationship between the two features. To make identifying the potential relationships easier, we have also added the regression lines.

In our case, we only plotted three features. That is because with 30,000 observations it can take quite some time to render the plot for all numeric columns, not to mention losing readability with so many small plots in one matrix. When using pairplots, we can also specify the hue argument to add a split for a category (such as sex, or education level).

We can also zoom into a relationship between two variables using a joint plot (`sns.jointplot`). It is a type of plot that combines a scatterplot to analyze the bivariate relationship and KDE plots or histograms to analyze the univariate distribution. In *Step 6*, we analyzed the relationship between age and limit balance.

In *Step 7*, we defined a function for plotting a heatmap representing the correlation matrix. In the function, we used a couple of operations to mask the upper triangular matrix and the diagonal (all diagonal elements of the correlation matrix are equal to 1). This way, the output is much easier to interpret. Using the annot argument of `sns.heatmap`, we could add the underlying numbers to the heatmap. However, we should only do so when the number of analyzed features is not too high. Otherwise, they will become unreadable.

To calculate the correlations, we used the `corr` method of a DataFrame, which by default calculates the **Pearson's correlation coefficient**. We did this only for numeric features. There are also methods for calculating the correlation of categorical features; we mention some of them in the *There's more...* section. Inspecting correlations is crucial, especially when using machine learning algorithms that assume linear independence of the features.

In *Step 8*, we used box plots to investigate the distribution of age by marital status and sex. A box plot (also called a box-and-whisker plot) presents the distribution of data in such a way that facilitates comparisons between levels of a categorical variable. A box plot presents the information about the distribution of the data using a 5-number summary:

- Median (50th percentile)—represented by the horizontal black line within the boxes.
- **Interquartile range (IQR)**—represented by the box. It spans the range between the first quartile (25th percentile) and the third quartile (75th percentile).
- The whiskers—represented by the lines stretched from the box. The extreme values of the whiskers (marked as horizontal lines) are defined as the first quartile – 1.5 IQR and the third quartile + 1.5 IQR.

We can use the box plots to gather the following insights about our data:

- The points marked outside of the whiskers can be considered outliers. This method is called **Tukey's fences** and is one of the simplest outlier detection techniques. In short, it assumes that observations lying outside of the $[Q1 - 1.5 \text{ IQR}, Q3 + 1.5 \text{ IQR}]$ range are outliers.

- The potential skewness of the distribution. A right-skewed (positive skewness) distribution can be observed when the median is closer to the lower bound of the box, and the upper whisker is longer than the lower one. Vice versa for the left-skewed distributions. *Figure 13.13* illustrates this.

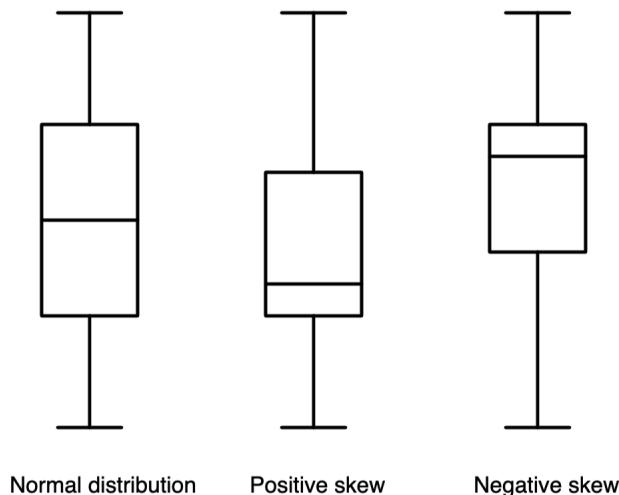


Figure 13.13: Determining the skewness of distribution using box plots

In *Step 9*, we used violin plots to investigate the distribution of the limit balance feature per education level and sex. We created them by using `sns.violinplot`. We indicated the education level with the `x` argument. Additionally, we set `hue="sex"` and `split=True`. By doing so, each half of the violin represented a different sex.

In general, violin plots are very similar to box plots and we can find the following information in them:

- The median, represented by a white dot.
- The interquartile range, represented as the black bar in the center of a violin.
- The lower and upper adjacent values, represented by the black lines stretched from the bar. The lower adjacent value is defined as the first quartile - 1.5 IQR, while the upper one is defined as the third quartile + 1.5 IQR. Again, we can use the adjacent values as a simple outlier detection technique.

Violin plots are a combination of a box plot and a KDE plot. A definite advantage of a violin plot over a box plot is that the former enables us to clearly see the shape of the distribution. This is especially useful when dealing with multimodal distributions (distributions with multiple peaks), such as the limit balance violin in the *Graduate school* education category.

In the last two steps, we investigated the distribution of the target variable (default) per sex and education. In the first case, we used `sns.countplot` to display the count of occurrences of both possible outcomes for each sex. In the second case, we opted for a different approach. We wanted to plot the percentage of defaults per education level, as comparing percentages between groups is easier than comparing nominal values. To do so, we first grouped by education level, selected the variable of interest, calculated the percentages per group (using the `value_counts(normalize=True)` method), unstacked (to remove multi-index), and generated a plot using the already familiar `plot` method.

There's more...

In this recipe, we introduced a range of possible approaches to investigate the data at hand. However, this requires many lines of code (quite a lot of them boilerplate) each time we want to carry out the EDA. Thankfully, there is a Python library that simplifies the process. The library is called `pandas_profiling` and with a single line of code, it generates a comprehensive summary of the dataset in the form of an HTML report.

To create a report, we need to run the following:

```
from pandas_profiling import ProfileReport  
profile = ProfileReport(df, title="Loan Default Dataset EDA")  
profile
```



We could also create a profile using the new (added by `pandas_profiling`) `profile_report` method of a `pandas` DataFrame.

For practical reasons, we might prefer to save the report as an HTML file and inspect it in a browser instead of the Jupyter notebook. We can easily do so using the following snippet:

```
profile.to_file("loan_default_eda.html")
```

The report is very exhaustive and contains a lot of useful information. Please see the following figure for an example.

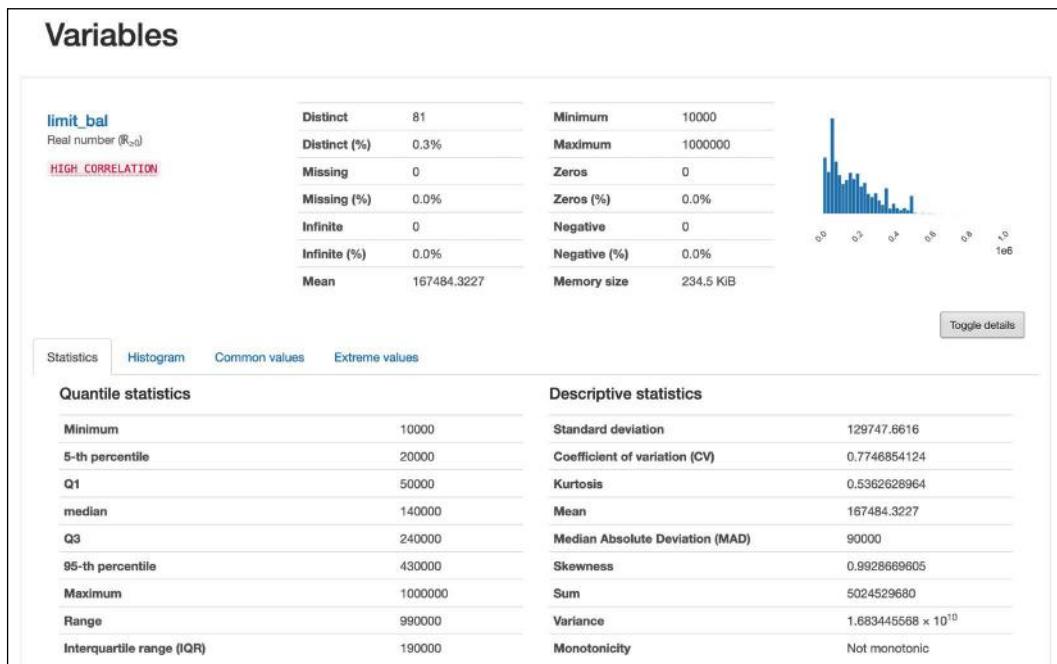


Figure 13.14: Example of a deep-dive into the limit balance feature

For brevity's sake, we will only discuss selected elements of the report:

- An overview giving information about the size of the DataFrame (number of features/rows, missing values, duplicated rows, memory size, breakdown per data type).
- Alerts warning us about potential issues with our data, including a high percentage of duplicated rows, highly correlated (and potentially redundant) features, features that have a high percentage of zero values, highly skewed features, etc.
- Different measures of correlation: Spearman's ρ , Pearson's r , Kendall's τ , Cramér's V , and Phik (ϕ_k). The last one is especially interesting, as it is a recently developed correlation coefficient that works consistently between categorical, ordinal, and interval variables. On top of that, it captures non-linear dependencies. Please see the *See also* section for a reference paper describing the metric.
- Detailed analysis of missing values.
- Detailed univariate analysis of each feature (more details are available by clicking *Toggle details* in the report).

pandas-profiling is the most popular auto-EDA tool in Python's vast ecosystem of libraries. However, it is definitely not the only one. You can also investigate the following:

- sweetviz—<https://github.com/fbdesignpro/sweetviz>
- autoviz—<https://github.com/AutoViML/AutoViz>
- dtale—<https://github.com/man-group/dtale>
- dataprep—<https://github.com/sfu-db/dataprep>
- lux—<https://github.com/lux-org/lux>

Each one of them approaches EDA a bit differently. Hence, it is best to explore them all and pick the tool that works best for your needs.

See also

For more information about Phik (φ_k), please see the following paper:

- Baak, M., Koopman, R., Snoek, H., & Klous, S. (2020). "A new correlation coefficient between categorical, ordinal and interval variables with Pearson characteristics." *Computational Statistics & Data Analysis*, 152, 107043. <https://doi.org/10.1016/j.csda.2020.107043>.

Splitting data into training and test sets

Having completed the EDA, the next step is to split the dataset into training and test sets. The idea is to have two separate datasets:

- Training set—on this part of the data, we train a machine learning model
- Test set—this part of the data was not seen by the model during training and is used to evaluate its performance

By splitting the data this way, we want to prevent overfitting. Overfitting is a phenomenon that occurs when a model finds too many patterns in data used for training and performs well only on that particular data. In other words, it fails to generalize to unseen data.

This is a very important step in the analysis, as doing it incorrectly can introduce bias, for example, in the form of data leakage. Data leakage can occur when, during the training phase, a model observes information to which it should not have access. We follow up with an example. A common scenario is that of imputing missing values with the feature's average. If we had done this before splitting the data, we would have also used data from the test set to calculate the average, introducing leakage. That is why the proper order would be to split the data into training and test sets first and then carry out the imputation, using the data observed in the training set. The same goes for setting up rules for identifying outliers.

Additionally, splitting the data ensures consistency, as unseen data in the future (in our case, new customers that will be scored by the model) will be treated in the same way as the data in the test set.

How to do it...

Execute the following steps to split the dataset into training and test sets:

1. Import the libraries:

```
import pandas as pd  
from sklearn.model_selection import train_test_split
```

2. Separate the target from the features:

```
X = df.copy()  
y = X.pop("default_payment_next_month")
```

3. Split the data into training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.2, random_state=42  
)
```

4. Split the data into training and test sets without shuffling:

```
X_train, X_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.2, shuffle=False  
)
```

5. Split the data into training and test sets with stratification:

```
X_train, X_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.2, stratify=y, random_state=42  
)
```

6. Verify that the ratio of the target is preserved:

```
print("Target distribution - train")  
print(y_train.value_counts(normalize=True).values)  
print("Target distribution - test")  
print(y_test.value_counts(normalize=True).values)
```

Running the snippet generates the following output:

```
Target distribution - train  
[0.77879167 0.22120833]  
Target distribution - test  
[0.77883333 0.22116667]
```

In both sets, the percentage of payment defaults is around 22.12%.

How it works...

After importing the libraries, we separated the target from the features using the pop method of a pandas DataFrame.

In *Step 3*, we showed how to do the most basic split. We passed `x` and `y` objects to the `train_test_split` function. Additionally, we specified the size of the test set, as a fraction of all observations. For reproducibility, we also specified the random state. We had to assign the output of the function to four new objects.

In *Step 4*, we took a different approach. By specifying `test_size=0.2` and `shuffle=False`, we assigned the first 80% of the data to the training set and the remaining 20% to the test set. This might come in handy when we want to preserve the order in which the observations are becoming available.

In *Step 5*, we also specified the stratification argument by passing the target variable (`stratify=y`). Splitting the data with stratification means that both the training and test sets will have an almost identical distribution of the specified variable. This parameter is very important when dealing with imbalanced data, such as in cases of fraud detection. If 99% of data is normal and only 1% covers fraudulent cases, a random split can result in the training set not having any fraudulent cases. That is why when dealing with imbalanced data, it is crucial to split it correctly.

In the very last step, we verified if the stratified train/test split resulted in the same ratio of defaults in both datasets. To verify that, we used the `value_counts` method of a pandas DataFrame.

In the rest of the chapter, we will use the data obtained from the stratified split.

There's more...

It is also common to split data into three sets: training, validation, and test. The **validation set** is used for frequent evaluation and tuning of the model's hyperparameters. Suppose we want to train a decision tree classifier and find the optimal value of the `max_depth` hyperparameter, which decides the maximum depth of the tree.

To do so, we can train the model multiple times using the training set, each time with a different value of the hyperparameter. Then, we can evaluate the performance of all these models, using the validation set. We pick the best model of those, and then ultimately evaluate its performance on the test set.

In the following snippet, we illustrate a possible way of creating a train-validation-test split, using the same `train_test_split` function:

```
import numpy as np

# define the size of the validation and test sets
VALID_SIZE = 0.1
TEST_SIZE = 0.2

# create the initial split - training and temp
X_train, X_temp, y_train, y_temp = train_test_split(
```

```
X, y,  
    test_size=(VALID_SIZE + TEST_SIZE),  
    stratify=y,  
    random_state=42  
)  
  
# calculate the new test size  
new_test_size = np.around(TEST_SIZE / (VALID_SIZE + TEST_SIZE), 2)  
  
# create the valid and test sets  
X_valid, X_test, y_valid, y_test = train_test_split(  
    X_temp, y_temp,  
    test_size=new_test_size,  
    stratify=y_temp,  
    random_state=42  
)
```

We basically ran `train_test_split` twice. What is important is that we had to adjust the sizes of the `test_size` input in such a way that the initially defined proportions (70-10-20) were preserved.

We also verify that everything went according to plan: that the size of the datasets corresponds to the intended split and that the ratio of defaults is the same in each set. We do so using the following snippet:

```
print("Percentage of data in each set ----")  
print(f"Train: {100 * len(X_train) / len(X):.2f}%")  
print(f"Valid: {100 * len(X_valid) / len(X):.2f}%")  
print(f"Test: {100 * len(X_test) / len(X):.2f}%")  
print("")  
print("Class distribution in each set ----")  
print(f"Train: {y_train.value_counts(normalize=True).values}")  
print(f"Valid: {y_valid.value_counts(normalize=True).values}")  
print(f"Test: {y_test.value_counts(normalize=True).values}")
```

Executing the snippet generates the following output:

```
Percentage of data in each set ----  
Train: 70.00%  
Valid: 9.90%  
Test: 20.10%  
  
Class distribution in each set ----  
Train: [0.77879899 0.22120101]  
Valid: [0.77878788 0.22121212]  
Test: [0.77880948 0.22119052]
```

We have indeed verified that the original dataset was split with the intended 70-10-20 ratio and that the distribution of defaults (target variable) was preserved due to stratification. Sometimes, we do not have enough data to split it into three sets, either because we do not have that many observations in general or because the data can be highly imbalanced, and we would remove valuable training samples from the training set. That is why practitioners often use a method called cross-validation, which we describe in the *Tuning hyperparameters using grid search and cross-validation* recipe.

Identifying and dealing with missing values

In most real-life cases, we do not work with clean, complete data. One of the potential problems we are bound to encounter is that of missing values. We can categorize missing values by the reason they occur:

- **Missing completely at random (MCAR)**—The reason for the missing data is unrelated to the rest of the data. An example could be a respondent accidentally missing a question in a survey.
- **Missing at random (MAR)**—The missingness of the data can be inferred from data in another column(s). For example, a missing response to a certain survey question can to some extent be determined conditionally by other factors such as sex, age, lifestyle, and so on.
- **Missing not at random (MNAR)**—When there is some underlying reason for the missing values. For example, people with very high incomes tend to be hesitant about revealing it.
- **Structurally missing data**—Often a subset of MNAR, the data is missing because of a logical reason. For example, when a variable representing the age of a spouse is missing, we can infer that a given person has no spouse.

Some machine learning algorithms can account for missing data, for example, decision trees can treat missing values as a separate and unique category. However, many algorithms either cannot do so or their popular implementations (such as the ones in `scikit-learn`) do not incorporate this functionality.



We should only impute features, not the target variable!

Some popular solutions to handling missing values include:

- Drop observations with one or more missing values—while this is the easiest approach, it is not always a good one, especially in the case of small datasets. We should also be aware of the fact that even if there is only a small fraction of missing values per feature, they do not necessarily occur for the same observations (rows), so the actual number of rows we might need to remove can be much higher. Additionally, in the case of data missing not at random, removing such observations from the analysis can introduce bias into the results.
- We can opt to drop the entire column (feature) if it is mostly populated with missing values. However, we need to be cautious as this can already be an informative signal for our model.
- Replace the missing values with a value far outside the possible range, so that algorithms such as decision trees can treat it as a special value, indicating missing data.

- In the case of dealing with time series, we can use forward-filling (take the last-known observation before the missing one), backward-filling (take the first-known observation after the missing one), or interpolation (linear or more advanced).
- **Hot-deck imputation**—in this simple algorithm, we first select one or more of the other features correlated with the one containing missing values. Then, we sort the rows of the dataset by these selected features. Lastly, we iterate over the rows from top to bottom and replace each missing value with the previous non-missing value in the same feature.
- Replace the missing values with an aggregate metric—for continuous data, we can use the mean (when there are no clear outliers in the data) or median (when there are outliers). In the case of categorical variables, we can use mode (the most common value in the set). Potential disadvantages of mean/median imputation include the reduction of variance in the dataset and distorting the correlations between the imputed features and the rest of the dataset.
- Replace the missing values with aggregate metrics calculated per group—for example, when dealing with body-related metrics, we can calculate the mean or median per sex, to more accurately replace the missing data.
- ML-based approaches—we can treat the considered feature as a target, and use complete cases to train a model and predict values for the missing observations.

In general, exploring the missing values is part of the EDA. We briefly touched upon it when analyzing the report generated with `pandas_profiling`. But we deliberately left it without much coverage until now, as it is crucial to carry out any kind of missing value imputation after the train/test split. Otherwise, we cause data leakage.

In this recipe, we show how to identify the missing values in our data and how to impute them.

Getting ready

For this recipe, we assume that we have the outputs of the stratified train/test split from the previous recipe, *Splitting data into training and test sets*.

How to do it...

Execute the following steps to investigate and deal with missing values in the dataset:

1. Import the libraries:

```
import pandas as pd
import missingno as msno
from sklearn.impute import SimpleImputer
```

2. Inspect the information about the DataFrame:

```
x.info()
```

Executing the snippet generates the following summary (abbreviated):

```
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 23 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   limit_bal        30000 non-null   int64  
 1   sex              29850 non-null   object  
 2   education        29850 non-null   object  
 3   marriage         29850 non-null   object  
 4   age              29850 non-null   float64 
 5   payment_status_sep 30000 non-null   object  
 6   payment_status_aug 30000 non-null   object  
 7   payment_status_jul 30000 non-null   object
```

Our dataset has more columns, however, the missing values are only present in the 4 columns visible in the summary. For brevity, we have not included the rest of the output.

3. Visualize the nullity of the DataFrame:

```
msno.matrix(X)
```

Running the line of code results in the following plot:

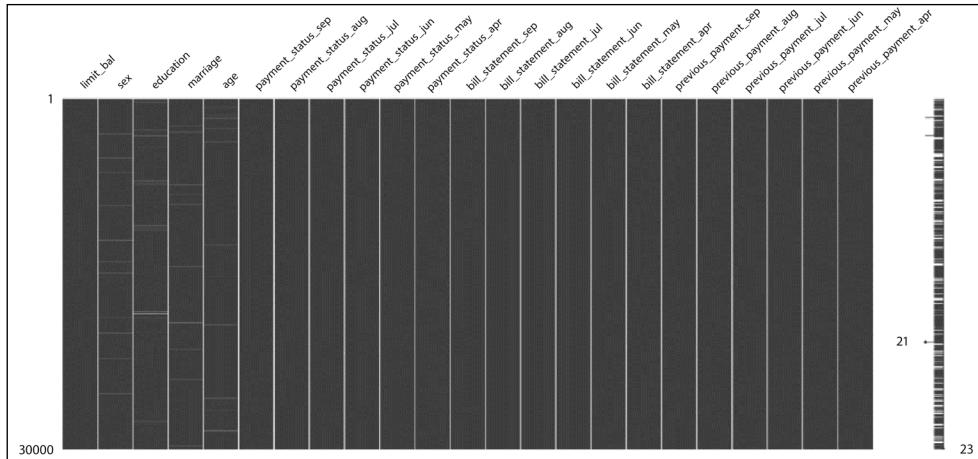


Figure 13.15: The nullity matrix plot of the loan default dataset

The white bars visible in the columns represent missing values. We should keep in mind that when working with large datasets with only a few missing values, those white bars might be quite difficult to spot.

The line on the right side of the plot describes the shape of data completeness. The two numbers indicate the maximum and minimum nullity in the dataset. When an observation has no missing values, the line will be at the maximum right position and have a value equal to the number of columns in the dataset (23 in this case). As the number of missing values starts to increase within an observation, the line moves towards the left. The nullity value of 21 indicates that there is a row with 2 missing values in it, as the maximum value for this dataset is 23 (the number of columns).

4. Define columns with missing values per data type:

```
NUM_FEATURES = ["age"]
CAT_FEATURES = ["sex", "education", "marriage"]
```

5. Impute numerical features:

```
for col in NUM_FEATURES:
    num_imputer = SimpleImputer(strategy="median")
    num_imputer.fit(X_train[[col]])
    X_train.loc[:, col] = num_imputer.transform(X_train[[col]])
    X_test.loc[:, col] = num_imputer.transform(X_test[[col]])
```

6. Impute categorical features:

```
for col in CAT_FEATURES:
    cat_imputer = SimpleImputer(strategy="most_frequent")
    cat_imputer.fit(X_train[[col]])
    X_train.loc[:, col] = cat_imputer.transform(X_train[[col]])
    X_test.loc[:, col] = cat_imputer.transform(X_test[[col]])
```

We can verify that neither the training nor test sets contain missing values using the `info` method.

How it works...

In *Step 1*, we imported the required libraries. Then, we used the `info` method of a pandas DataFrame to view information about the columns, such as their type and the number of non-null observations. The difference between the total number of observations and the non-null ones corresponds to the number of missing observations. Another way of inspecting the number of missing values per column is to run `X.isnull().sum()`.

Instead of imputing, we could also drop the observations (or even columns) containing missing values. To drop all rows containing any missing value, we could use `X_train.dropna(how="any", inplace=True)`. In our sample case, the number of missing values is not large, however, in a real-life dataset it can be considerable or the dataset might be too small for the analysts to be able to remove observations. Alternatively, we could also specify the `thresh` argument of the `dropna` method to indicate in how many columns an observation (row) needs to have missing values in order to be dropped from the dataset.

In *Step 3*, we visualized the nullity of the DataFrame, with the help of the `missingno` library.

In *Step 4*, we defined lists containing features we wanted to impute, one list per data type. The reason for this is the fact that numeric features are imputed using different strategies than categorical features. For basic imputation, we used the `SimpleImputer` class from `scikit-learn`.

In *Step 5*, we iterated over the numerical features (in this case, only the age feature), and used the median to replace the missing values. Inside the loop, we defined the imputer object with the correct strategy ("`median`"), fitted it to the given column of the training data, and transformed both the training and test data. This way, the median was estimated by using only the training data, preventing potential data leakage.



In this recipe, we used `scikit-learn` to deal with the imputation of missing values. However, we can also do this manually. To do so, for each column with any missing values (either in the training or test set), we need to calculate the given statistic (mean/median/mode) using the training set, for example, `age_median = X_train.age.median()`. Afterward, we need to use this median to fill in the missing values for the age column (in both the training and test sets) using the `fillna` method. We show how to do it in the Jupyter notebook available in the book's GitHub repository.

Step 6 is analogous to *Step 5*, where we used the same approach to iterate over categorical columns. The difference lies in the selected strategy—we used the most frequent value ("`most_frequent`") in the given column. This strategy can be used for both categorical and numerical features. In the latter case, it corresponds to the mode.

There's more...

There are a few more things worth mentioning when covering handling missing values.

More visualizations available in the `missingno` library

In this recipe, we have already covered the nullity matrix representation of missing values in a dataset. However, the `missingno` library offers a few more helpful visualizations:

- `msno.bar`—generates a bar chart representing the nullity in each column. Might be easier to quickly interpret than the nullity matrix.
- `msno.heatmap`—visualizes the nullity correlation, that is, how strongly the presence/absence of one feature impacts the presence of another. The interpretation of the nullity correlation is very similar to the standard Pearson's correlation coefficient. It ranges from -1 (when one feature occurs, then the other one certainly does not) through 0 (features appearing or not appearing have no effect on each other) to 1 (if one feature occurs, then the other one certainly does too).
- `msno.dendrogram`—allows us to better understand the correlations between variable completion. Under the hood, it uses hierarchical clustering to bin features against one another by their nullity correlation.

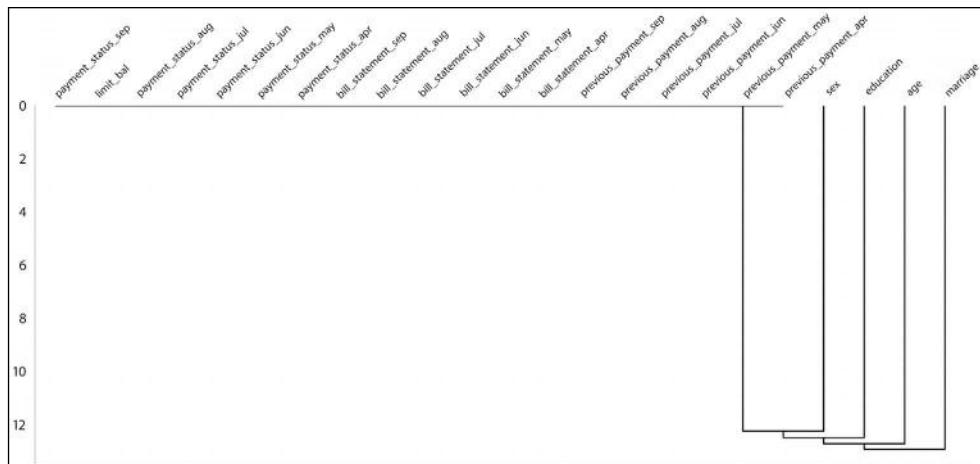


Figure 13.16: Example of the nullity dendrogram

To interpret the figure, we need to analyze it from a top-down perspective. First, we should look at cluster leaves, which are linked at a distance of zero. Those fully predict each other's presence, that is, one feature might always be missing when the other is present, or they might always both be present or missing, and so on. Cluster leaves with a split close to zero predict each other very well.

In our case, the dendrogram links together the features that are present in every observation. We know this for certain, as we have introduced the missing observations by design in only four features.

ML-based approaches to imputing missing values

In this recipe, we mentioned how to impute missing values. Approaches such as replacing the missing values with one large value or the mean/median/mode are called **single imputation approaches**, as they replace missing values with one specific value. On the other hand, there are also **multiple imputation approaches**, and one of those is **Multiple Imputation by Chained Equations (MICE)**.

In short, the algorithm runs multiple regression models, and each missing value is determined conditionally on the basis of the non-missing data points. A potential benefit of using an ML-based approach to imputation is the reduction of bias introduced by single imputation. The MICE algorithm is available in `scikit-learn` under the name of `IterativeImputer`.

Alternatively, we could use the **nearest neighbors imputation** (implemented in `scikit-learn`'s `KNNImputer`). The underlying assumption of the KNN imputation is that a missing value can be approximated by the values of the same feature coming from the observations that are closest to it. The closeness to the other observations is determined using other features and some form of a distance metric, for example, the Euclidean distance.

As the algorithm uses KNN, it comes with some of its drawbacks:

- Requires tuning of the k hyperparameter for best performance
- We need to scale the data and preprocess categorical features
- We need to pick an appropriate distance metric (especially in cases when we have a mix of categorical and numerical features)
- The algorithm is sensitive to outliers and noise in data
- Can be computationally expensive as it requires calculating the distances between every pair of observations

Another of the available ML-based algorithms is called **MissForest** (available in the `missForest` library). Without going into too much detail, the algorithm starts by filling in the missing values with the median or mode imputation. Then, it trains a Random Forest model to predict the feature that is missing using the other known features. The model is trained using the observations for which we know the values of the target (so the ones that were not imputed in the first step) and then makes predictions for the observations with the missing feature. In the next step, the initial median/mode prediction is replaced with the one coming from the RF model. The process of looping through the missing data points is repeated several times, and each iteration tries to improve upon the previous one. The algorithm stops when some stopping criterion is met or we exhaust the allowed number of iterations.

Advantages of MissForest:

- Can handle missing values in both numeric and categorical features
- Does not require data preprocessing (such as scaling)
- Robust to noisy data, as Random Forest makes little to no use of uninformative features
- Non-parametric—it does not make assumptions about the relationship between the features (MICE assumes linearity)
- Can leverage non-linear and interaction effects between features to improve imputation performance

Disadvantages of MissForest:

- Imputation time increases with the number of observations, features, and the number of features containing missing values
- Similar to Random Forest, not very easy to interpret
- It is an algorithm and not a model object we can store somewhere (for example, as a pickle file) and reuse whenever we need to impute missing values

See also

Additional resources are available here:

- Azur, M. J., Stuart, E. A., Frangakis, C., & Leaf, P. J. (2011). “Multiple imputation by chained equations: what is it and how does it work?” *International Journal of Methods in Psychiatric Research*, 20(1), 40-49. <https://doi.org/10.1002/mpr.329>.

- Buck, S. F. (1960). “A method of estimation of missing values in multivariate data suitable for use with an electronic computer.” *Journal of the Royal Statistical Society: Series B (Methodological)*, 22(2), 302–306. <https://www.jstor.org/stable/2984099>.
- Stekhoven, D. J. & Bühlmann, P. (2012). “MissForest—non-parametric missing value imputation for mixed-type data.” *Bioinformatics*, 28(1), 112–118.
- van Buuren, S. & Groothuis-Oudshoorn, K. (2011). “MICE: Multivariate Imputation by Chained Equations in R.” *Journal of Statistical Software* 45 (3): 1–67.
- Van Buuren, S. (2018). *Flexible Imputation of Missing Data*. CRC press.
- `miceforest`—a Python library for fast, memory-efficient MICE with LightGBM.
- `missingpy`—a Python library containing the implementation of the MissForest algorithm.

Encoding categorical variables

In the previous recipes, we have seen that some features are categorical variables (originally represented as either `object` or `category` data types). However, most machine learning algorithms work exclusively with numeric data. That is why we need to encode categorical features into a representation compatible with the ML models.

The first approach to encoding categorical features is called **label encoding**. In this approach, we replace the categorical values of a feature with distinct numeric values. For example, with three distinct classes, we use the following representation: `[0, 1, 2]`.



This is already very similar to the outcome of converting to the `category` data type in pandas. Let's assume we have a DataFrame called `df_cat`, which has a feature called `feature_1`. This feature is encoded as the `category` data type. We can then access the codes of the categories by running `df_cat["feature_1"].cat.codes`. Additionally, we can recover the mapping by running `dict(zip(df_cat["feature_1"].cat.codes, df_cat["feature_1"]))`. We can also use the `pd.factorize` function to arrive at a very similar representation.

One potential issue with label encoding is that it introduces a relationship between the categories, while often there is none. In a three-classes example, the relationship looks as follows: $0 < 1 < 2$. This does not make much sense if the categories are, for example, countries. However, this can work for features that represent some kind of order (ordinal variables). For example, label encoding could work well with a rating of service received, on a scale of Bad-Neutral-Good.

To overcome the preceding problem, we can use **one-hot encoding**. In this approach, for each category of a feature, we create a new column (sometimes called a dummy variable) with binary encoding to denote whether a particular row belongs to this category. A potential drawback of this method is that it significantly increases the dimensionality of the dataset (**curse of dimensionality**). First, this can increase the risk of overfitting, especially when we do not have that many observations in our dataset. Second, a high-dimensional dataset can be a significant problem for any distance-based algorithm (for example, k-Nearest Neighbors), as—on a very high level—a large number of dimensions causes all the observations to appear equidistant from each other. This can naturally render the distance-based models useless.



Another thing we should be aware of is that creating dummy variables introduces a form of redundancy to the dataset. In fact, if a feature has three categories, we only need two dummy variables to fully represent it. That is because if an observation is neither of the two, it must be the third one. This is often referred to as the **dummy-variable trap**, and it is best practice to always remove one column (known as the reference value) from such an encoding. This is especially important in unregularized linear models.

Summing up, we should avoid label encoding as it introduces false order to the data, which can lead to incorrect conclusions. Tree-based methods (decision trees, Random Forest, and so on) can work with categorical data and label encoding. However, one-hot encoding is the natural representation of categorical features for algorithms such as linear regression, models calculating distance metrics between features (such as k-means clustering or k-Nearest Neighbors), or **Artificial Neural Networks (ANN)**.

Getting ready

For this recipe, we assume that we have the outputs of the imputed training and test sets from the previous recipe, *Identifying and dealing with missing values*.

How to do it...

Execute the following steps to encode categorical variables with label encoding and one-hot encoding:

1. Import the libraries:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer
```

2. Use Label Encoder to encode a selected column:

```
COL = "education"

X_train_copy = X_train.copy()
X_test_copy = X_test.copy()

label_enc = LabelEncoder()
label_enc.fit(X_train_copy[COL])
X_train_copy.loc[:, COL] = label_enc.transform(X_train_copy[COL])
X_test_copy.loc[:, COL] = label_enc.transform(X_test_copy[COL])

X_test_copy[COL].head()
```

Running the snippet generates the following preview of the transformed column:

```
6907    3  
24575    0  
26766    3  
2156     0  
3179     3  
Name: education, dtype: int64
```

We created a copy of `X_train` and `X_test`, just to show how to work with `LabelEncoder`, but we do not want to modify the actual DataFrames we intend to use later.



We can access the labels stored within the fitted `LabelEncoder` by using the `classes_` attribute.

3. Select categorical features for one-hot encoding:

```
cat_features = X_train.select_dtypes(include="object") \  
                  .columns \  
                  .to_list()  
  
cat_features
```

We will apply one-hot encoding to the following columns:

```
['sex', 'education', 'marriage', 'payment_status_sep', 'payment_status_  
aug', 'payment_status_jul', 'payment_status_jun', 'payment_status_may',  
'payment_status_apr']
```

4. Instantiate the `OneHotEncoder` object:

```
one_hot_encoder = OneHotEncoder(sparse=False,  
                                 handle_unknown="error",  
                                 drop="first")
```

5. Create the column transformer using the one-hot encoder:

```
one_hot_transformer = ColumnTransformer(  
    [("one_hot", one_hot_encoder, cat_features)],  
    remainder="passthrough",  
    verbose_feature_names_out=False  
)
```

6. Fit the transformer:

```
one_hot_transformer.fit(X_train)
```

Executing the snippet prints the following preview of the column transformer:

```

ColumnTransformer
ColumnTransformer(remainder='passthrough',
                  transformers=[('one_hot',
                                 OneHotEncoder(drop='first', sparse=False),
                                 ['sex', 'education', 'marriage',
                                  'payment_status_sep', 'payment_status_aug',
                                  'payment_status_jul', 'payment_status_jun',
                                  'payment_status_may',
                                  'payment_status_apr'))],
                  verbose_feature_names_out=False)
    ▶   one_hot
        ▼   OneHotEncoder
            OneHotEncoder(drop='first', sparse=False)
    ▶   remainder
        ▼   passthrough
            passthrough

```

Figure 13.17: Preview of the column transformer with one-hot encoding

7. Apply the transformations to both training and test sets:

```

col_names = one_hot_transformer.get_feature_names_out()

X_train_ohe = pd.DataFrame(
    one_hot_transformer.transform(X_train),
    columns=col_names,
    index=X_train.index
)

X_test_ohe = pd.DataFrame(one_hot_transformer.transform(X_test),
                           columns=col_names,
                           index=X_test.index)

```

As we have mentioned before, one-hot encoding comes with the potential disadvantage of increasing the dimensionality of the dataset. In our case, we started with 23 columns. After applying one-hot encoding, we ended up with 72 columns.

How it works...

First, we imported the required libraries. In the second step, we selected the column we wanted to encode using label encoding, instantiated the `LabelEncoder`, fitted it to the training data, and transformed both the training and the test data. We did not want to keep the label encoding, and for that reason we operated on copies of the `DataFrames`.



We demonstrated using label encoding as it is one of the available options, however, it comes with quite serious drawbacks. So in practice, we should refrain from using it. Additionally, `scikit-learn`'s documentation warns us with the following statement: *This transformer should be used to encode target values, i.e. `y`, and not the input `X`.*

In *Step 3*, we started the preparations for one-hot encoding by creating a list of all the categorical features. We used the `select_dtypes` method to select all features with the `object` data type.

In *Step 4*, we created an instance of `OneHotEncoder`. We specified that we did not want to work with a sparse matrix (a special kind of data type, suitable for storing matrices with a very high percentage of zeros), we dropped the first column per feature (to avoid the dummy variable trap), and we specified what to do if the encoder finds an unknown value while applying the transformation (`handle_unknown='error'`).

In *Step 5*, we defined the `ColumnTransformer`, which is a convenient approach to applying the same transformation (in this case, the one-hot encoder) to multiple columns. We passed a list of steps, where each step was defined by a tuple. In this case, it was a single tuple with the name of the step ("one_hot"), the transformation to be applied, and the features to which we wanted to apply the transformation.

When creating the `ColumnTransformer`, we also specified another argument, `remainder="passthrough"`, which has effectively fitted and transformed only the specified columns, while leaving the rest intact. The default value for the `remainder` argument was "drop", which dropped the unused columns. We also specified the value of the `verbose_feature_names_out` argument as `False`. This way, when we use the `get_feature_names_out` method later, it will not prefix all feature names with the name of the transformer that generated that feature.

If we had not changed it, some features would have the "one_hot_" prefix, while the others would have "remainder_".

In *Step 6*, we fitted the column transformer to the training data using the `fit` method. Lastly, we applied the transformations using the `transform` method to both training and test sets. As the `transform` method returns a `numpy` array instead of a `pandas DataFrame`, we had to convert them ourselves. We started by extracting the names of the features using the `get_feature_names_out`. Then, we created a `pandas DataFrame` using the transformed features, the new column names, and the old indices (to keep everything in order).



Similar to handling missing values or detecting outliers, we fit all the transformers (including one-hot encoding) to the training data only, and then we apply the transformations to both training and test sets. This way, we avoid potential data leakage.

There's more...

We would like to mention a few more things regarding the encoding of categorical variables.

Using pandas for one-hot encoding

Alternatively to `scikit-learn`, we can use `pd.get_dummies` for one-hot encoding categorical features. The example syntax looks like the following:

```
pd.get_dummies(X_train, prefix_sep="_", drop_first=True)
```

It's good to know this alternative approach, as it can be easier to work with (column names are automatically accounted for), especially when creating a quick Proof of Concept (PoC). However, when productionizing the code, the best approach would be to use the `scikit-learn` variant and create the dummy variables within a pipeline.

Specifying possible categories for OneHotEncoder

When creating `ColumnTransformer`, we could have additionally provided a list of possible categories for all the considered features. A simplified example follows:

```
one_hot_encoder = OneHotEncoder(  
    categories=[[ "Male", "Female", "Unknown" ]],  
    sparse=False,  
    handle_unknown="error",  
    drop="first"  
)  
  
one_hot_transformer = ColumnTransformer(  
    [("one_hot", one_hot_encoder, ["sex"])]  
)  
  
one_hot_transformer.fit(X_train)  
one_hot_transformer.get_feature_names_out()
```

Executing the snippet returns the following:

```
array(['one_hot__sex_Female', 'one_hot__sex_Unknown'], dtype=object)
```

By passing a list (of lists) containing possible categories for each feature, we are taking into account the possibility that the specific value does not appear in the training set, but might appear in the test set (or as part of the batch of new observations in the production environment). If this were the case, we would run into errors.

In the preceding code block, we added an extra category called "Unknown" to the column representing sex. As a result, we will end up with an extra "dummy" column for that category. The male category was dropped as the reference one.

Category Encoders library

Aside from using `pandas` and `scikit-learn`, we can also use another library called `Category Encoders`. It belongs to a set of libraries compatible with `scikit-learn` and provides a selection of encoders using a similar fit-transform approach. That is why it is also possible to use them together with `ColumnTransformer` and `Pipeline`.

We show an alternative implementation of the one-hot encoder.

Import the library:

```
import category_encoders as ce
```

Create the encoder object:

```
one_hot_encoder_ce = ce.OneHotEncoder(use_cat_names=True)
```

Additionally, we could specify an argument called `drop_invariant`, to indicate that we want to drop columns with no variance, so for example columns filled with only one distinct value. This could help with reducing the number of features.

Fit the encoder, and transform the data:

```
one_hot_encoder_ce.fit(X_train)
X_train_ce = one_hot_encoder_ce.transform(X_train)
```

This implementation of the one-hot encoder automatically encodes only the columns containing strings (unless we specify only a subset of categorical columns by passing a list to the `cols` argument). By default, it also returns a pandas DataFrame (in comparison to the numpy array, in the case of scikit-learn's implementation) with the adjusted column names. The only drawback of this implementation is that it does not allow for dropping the one redundant dummy column of each feature.

A warning about one-hot encoding and decision tree-based algorithms

While regression-based models can naturally handle the OR condition of one-hot-encoded features, the same is not that simple with decision tree-based algorithms. In theory, decision trees are capable of handling categorical features without the need for encoding.

However, its popular implementation in scikit-learn still requires all features to be numerical. Without going into too much detail, such an approach favors continuous numerical features over one-hot-encoded dummies, as a single dummy can only bring a fraction of the total feature information into the model. A possible solution is to use either a different kind of encoding (label/target encoding) or an implementation that handles categorical features, such as Random Forest in the h2o library or the LightGBM model.

Fitting a decision tree classifier

A decision tree classifier is a relatively simple yet very important machine learning algorithm used for both regression and classification problems. The name comes from the fact that the model creates a set of rules (for example, `if x_1 > 50 and x_2 < 10 then y = 'default'`), which taken together can be visualized in the form of a tree. The decision trees segment the feature space into a number of smaller regions, by repeatedly splitting the features at a certain value. To do so, they use a **greedy algorithm** (together with some heuristics) to find a split that minimizes the combined impurity of the children nodes. The impurity in classification tasks is measured using the Gini impurity or entropy, while for regression problems the trees use the mean squared error or the mean absolute error as the metric.

In the case of a binary classification problem, the algorithm tries to obtain nodes that contain as many observations from one class as possible, thus minimizing the impurity. The prediction in a terminal node (leaf) is made on the basis of mode in the case of classification, and mean for regression problems.



Decision trees are a base for many complex algorithms, such as Random Forest, Gradient Boosted Trees, XGBoost, LightGBM, CatBoost, and so on.

The advantages of decision trees include the following:

- Easily visualized in the form of a tree—high interpretability
- Fast training and prediction stages
- A relatively small number of hyperparameters to tune
- Support numerical and categorical features
- Can handle non-linearity in data
- Can be further improved with feature engineering, though there is no explicit need to do so
- Do not require scaling or normalization of features
- Incorporate their version of feature selection by choosing the features on which to split the sample
- Non-parametric model—no assumptions about the distribution of the features/target

On the other hand, the disadvantages of decision trees include the following:

- Instability—the trees are very sensitive to the noise in input data. A small change in the data can change the model significantly.
- Overfitting—if we do not provide maximum values or stopping criteria, the trees tend to grow very deep and do not generalize well.
- The trees can only interpolate, but not extrapolate—they make constant predictions for observations that lie beyond the boundary region established on the feature space of the training data.
- The underlying greedy algorithm does not guarantee the selection of a globally optimal decision tree.
- Class imbalance can lead to biased trees.
- Information gain (a decrease in entropy) in a decision tree with categorical variables results in a biased outcome for features with a higher number of categories.

Getting ready

For this recipe, we assume that we have the outputs of the one-hot-encoded training and test sets from the previous recipe, *Encoding categorical variables*.

How to do it...

Execute the following steps to fit a decision tree classifier:

1. Import the libraries:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn import metrics

from chapter_13_utils import performance_evaluation_report
```

In this recipe and the following ones, we will be using the `performance_evaluation_report` helper function. It plots useful metrics (confusion matrix, ROC curve) used for evaluating binary classification models. Also, it returns a dictionary containing more metrics, which we cover in the *How it works...* section.

2. Create the instance of the model, fit it to the training data, and create predictions:

```
tree_classifier = DecisionTreeClassifier(random_state=42)
tree_classifier.fit(X_train_ohe, y_train)
y_pred = tree_classifier.predict(X_test_ohe)
```

3. Evaluate the results:

```
LABELS = ["No Default", "Default"]
tree_perf = performance_evaluation_report(tree_classifier,
                                         X_test_ohe,
                                         y_test, labels=LABELS,
                                         show_plot=True)
```

Executing the snippet generates the following plot:

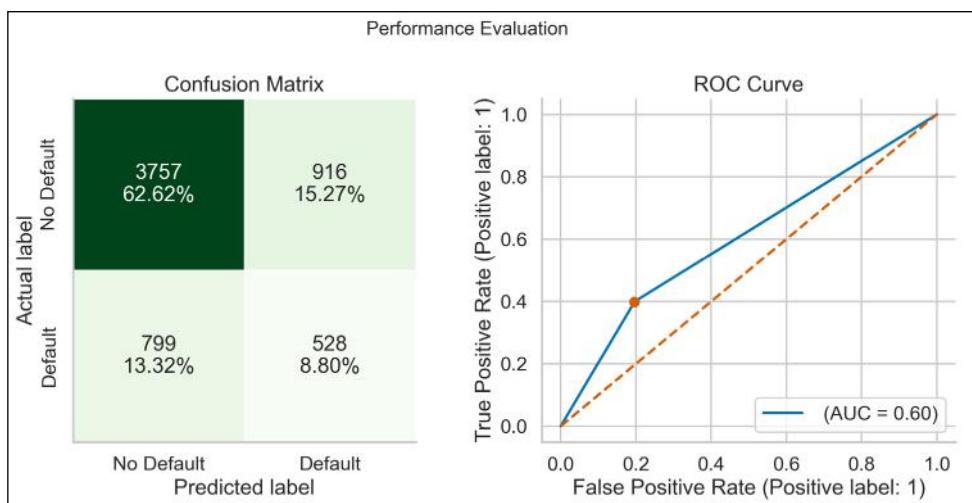


Figure 13.18: The performance evaluation report of the fitted decision tree classifier

The `tree_perf` object is a dictionary containing more relevant metrics, which can further help us with evaluating the performance of our model. We present those metrics below:

```
{'accuracy': 0.7141666666666666,
'precision': 0.3656509695290859,
'recall': 0.39788997739261495,
'specificity': 0.8039803124331265,
'f1_score': 0.3810898592565861,
'cohens_kappa': 0.1956931046277427,
'matthews_corr_coeff': 0.1959883714391891,
'roc_auc': 0.601583581287813,
'pr_auc': 0.44877724015824927,
'average_precision': 0.2789754297204212}
```

For more insights into the interpretation of the evaluation metrics, please refer to the *How it works...* section.

4. Plot the first few levels of the fitted decision tree:

```
plot_tree(tree_classifier, max_depth=3, fontsize=10)
```

Executing the snippet generates the following plot:

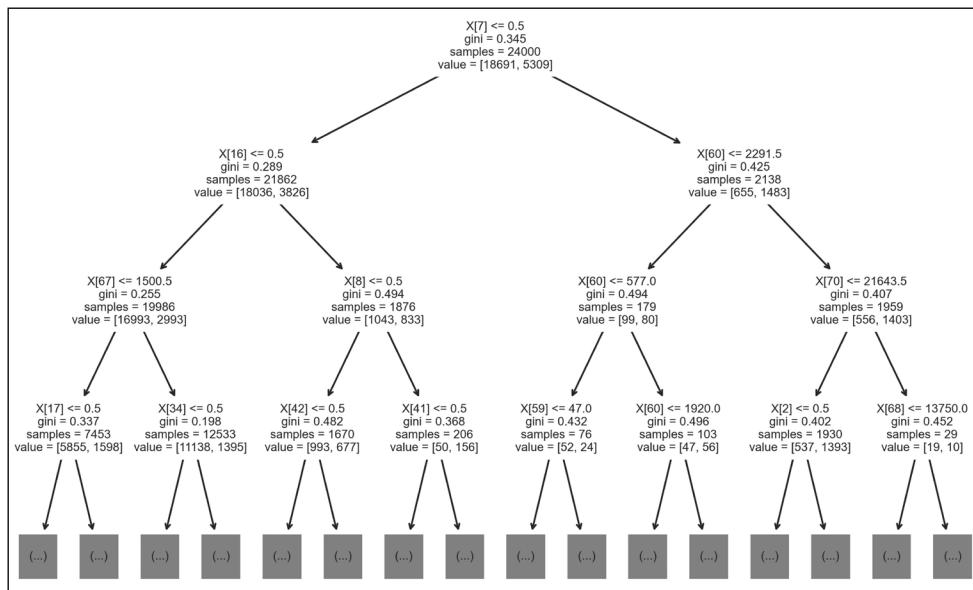


Figure 13.19: The fitted decision tree, capped at a max depth of 3

Using the one-liner, we can already visualize quite a lot of information. We decided to plot only the 3 levels of the decision tree, as the fitted tree actually reached the depth of 44 levels. As we have mentioned, not restricting the `max_depth` hyperparameter can lead to such cases, which are also very likely to overfit.

In the tree, we can see the following information:

- Which feature is used to split the tree and at which value. Unfortunately, with the default settings, we only see the column number instead of the feature's name. We will fix that soon.
 - The value of the Gini impurity.
 - The number of samples in each node/leaf.
 - The number of observations of each class within the node/leaf.

We can add more information to the plot with a few additional arguments of the `plot_tree` function:

```
plot_tree(  
    tree_classifier,  
    max_depth=2,  
    feature_names=X_train_ohe.columns,  
    class_names=["No default", "Default"],  
    rounded=True,  
    filled=True,  
    fontsize=10  
)
```

Executing the snippet generates the following plot:

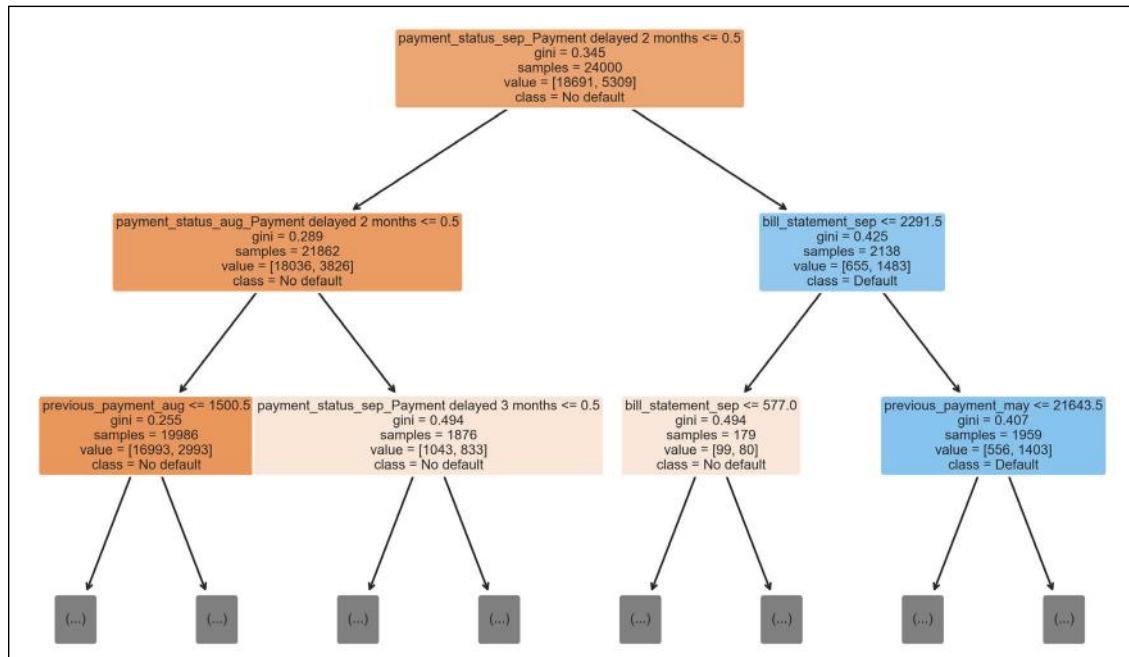


Figure 13.20: The fitted decision tree, capped at a max depth of 2.

In *Figure 13.20*, we see some additional information:

- The name of the feature used for creating the split
- The name of the class dominating in each node/leaf

Visualizing decision trees has many benefits. First, we can gain insights into which features are used for creating the model (a possible measure of feature importance) and what values were used to create the splits. Provided that the features have clear interpretation, this could work as a form of a sanity check to see if our initial hypotheses about the data and the considered problem came true and are aligned with common sense or domain knowledge. It could also help with presenting a clear and coherent story to the business stakeholders, who can quite easily understand such a simple representation of the model. We will discuss the feature importance and model explainability in depth in the following chapter.

How it works...

In *Step 2*, we used the typical `scikit-learn` approach to training a machine learning model. First, we created the object of the `DecisionTreeClassifier` class (using all the default settings and a fixed random state). Then, we fitted the model to the training data (we needed to pass both the features and the target) using the `fit` method. Lastly, we obtained the predictions by using the `predict` method.



Using the `predict` method results in an array of predicted classes (in this case, it is either a 0 or a 1). However, there are cases when we are interested in the assigned probabilities or scores. To obtain those, we can use the `predict_proba` method, which returns an array of size `n_test_observations` by `n_classes`. Each row contains all the possible class probabilities (they sum up to 1). In the case of binary classification, the `predict` method automatically assigns a positive class when the corresponding probability is above 50%.

In *Step 3*, we evaluated the performance of the model. We used a custom function to display all the results. We will not go deeper into its specifics, as it is quite standard and is built using functions from the `metrics` module of `scikit-learn`. For a detailed description of the function, please refer to the accompanying GitHub repository.

The **confusion matrix** summarizes all possible combinations of the predicted values as opposed to the actual target. The possible values are as follows:

- **True positive (TP):** The model predicts a default, and the client defaulted
- **False positive (FP):** The model predicts a default, but the client did not default
- **True negative (TN):** The model predicts a good customer, and the client did not default
- **False negative (FN):** The model predicts a good customer, but the client defaulted

In the scenarios presented above, we assumed that default is represented by the positive class. It does not mean that the outcome (client defaulting) is good or positive, just that an event occurred. Most frequently, the majority class is the “uninteresting” case and is assigned the negative label. This is a typical convention used in data science projects.

Using the presented values, we can further build multiple evaluation criteria:

- **Accuracy** [expressed as $(TP + TN) / (TP + FP + TN + FN)$]—measures the model’s overall ability to correctly predict the class of the observation.
- **Precision** [expressed as $TP / (TP + FP)$]—measures what fraction of all predictions of the positive class (in our case, the default) indeed were positive. In our project, it answers the question: *Out of all predictions of default, how many clients actually defaulted?* Or in other words: *When the model predicts default, how often is it correct?*
- **Recall** [expressed as $TP / (TP + FN)$]—measures what fraction of all positive cases were predicted correctly. Also called sensitivity or the true positive rate. In our case, it answers the question: *What fraction of all observed defaults did we predict correctly?*
- **F-1 Score**—a harmonic average of precision and recall. The reason for using the harmonic mean instead of arithmetic average is that it takes into account the harmony (similarity) between the two scores. Thus, it punishes extreme outcomes and discourages highly unequal values. For example, a classifier with precision = 1 and recall = 0 would score a 0.5 using a simple average, but a 0 when using the harmonic mean.
- **Specificity** [expressed as $TN / (TN + FP)$]—measures what fraction of negative cases (clients without a default) actually did not default. A helpful way of thinking about specificity is to consider it the recall of the negative class.

Understanding the subtleties behind these metrics is very important for the correct evaluation of the model’s performance. Accuracy can be highly misleading in the case of class imbalance. Imagine a case when 99% of data is not fraudulent and only 1% is fraudulent. Then, a naïve model classifying each observation as non-fraudulent achieves 99% accuracy, while it is actually worthless. That is why, in such cases, we should refer to precision or recall:

- When we try to achieve as high precision as possible, we will get fewer false positives, at the cost of more false negatives. We should optimize for precision when the cost of a false positive is high, for example, in spam detection.
- When optimizing for recall, we will achieve fewer false negatives, at the cost of more false positives. We should optimize for recall when the cost of a false negative is high, for example, in fraud detection.



There is no one-size-fits-all rule about which metric is best. The metric that we try to optimize for should be selected based on the use case.

The second plot contains the **Receiver Operating Characteristic (ROC)** curve. The ROC curve presents a trade-off between the true positive rate (TPR, recall) and the false positive rate (FPR, which is equal to 1 minus specificity) for different probability thresholds. A probability threshold determines the predicted probability above which we decide that the observation belongs to the positive class (by default, it is 50%).

An ideal classifier would have a false positive rate of 0 and a true positive rate of 1. Thus, the sweet spot in the ROC plot is the (0,1) point in the plot. A skillful model's curve would be as close to it as possible. On the other hand, a model with no skill will have a line close to the diagonal (45°) line. To better understand the ROC curve, please consider the following:

- Let's assume that we pick the decision threshold to be 0, that is, all observations are classified as defaults. This leads to two conclusions. First, no actual defaults are predicted as the negative class (false negatives), which means that the true positive rate (recall) is 1. Second, no good customers are classified as such (true negatives), which means that the false positive rate is also 1. This corresponds to the top-right corner of the ROC curve.
- Let's move to the other extreme and assume that the decision threshold is 1, that is, all customers are classified as good customers (no default, that is, the negative class). As there are no positive predictions at all, this leads to the following conclusions. First, there are no true positives ($TPR = 0$). Second, there are no false positives ($FPR = 0$). Such a scenario corresponds to the bottom left of the curve.
- As such, all the points on the curve correspond to the scores of a classifier for thresholds between the two extremes (0 and 1). The curve should approach the ideal point, in which the true positive rate is 1 and the false positive rate is 0. That is, no defaulting client is classified as a good customer and no good customer is classified as likely to default. In other words, a perfect classifier.
- If the performance approaches the diagonal line, the model is classifying roughly as many defaulting and non-defaulting customers as defaulting. In other words, this would be a classifier as good as random guessing.



A model with a curve below the diagonal line is possible and is actually better than the “no-skill” one, as its predictions can be simply inverted to obtain better performance.

To summarize the performance of a model with one number, we can look at the **area under the ROC curve** (AUC). It is an aggregate measure of performance across all possible decision thresholds. It is a metric with values between 0 and 1 and it tells us how much the model is capable of distinguishing between the classes. A model with an AUC of 0 is always wrong, while a model with an AUC of 1 is always correct. An AUC of 0.5 indicates a model with no skill that is pretty much equal to random guessing.

We can interpret the AUC in probabilistic terms. In short, it indicates how well the probabilities from the positive classes are separated from the negative classes. AUC represents the probability that a model ranks a random positive observation more highly than a random negative one.

An example might make it a bit easier to understand. Let's assume we have predictions obtained from some model, ranked in ascending order by their score/probability. *Figure 13.21* illustrates this. An AUC of 75% means that if we take one random positive observation and one random negative observation, with a 75% probability they will be ordered in the correct way, that is, the random positive example is to the right of the random negative example.

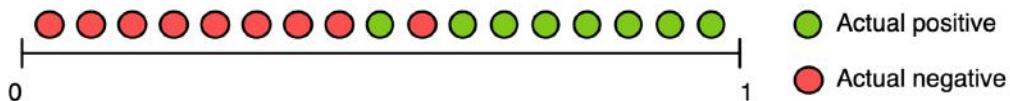


Figure 13.21: Model's output ordered by predicted score/probability



In practice, we may use the ROC curve to select a threshold that results in an appropriate balance between false positives and false negatives. Furthermore, AUC is a good metric to compare the difference in performance between various models.

In the last step, we visualized the decision tree using the `plot_tree` function.

There's more...

We have already covered the basics of using an ML model (in our case, a decision tree) to solve a binary classification task. We have also gone through the most popular classification evaluation metrics. However, there are still a few interesting topics to at least mention.

Diving deeper into classification evaluation metrics

One of the metrics we have covered quite extensively was the ROC curve. One issue with it is that it loses its credibility when it comes to evaluating the performance of the model when we are dealing with (severe) class imbalance. In such cases, we should use another curve—the **Precision-Recall curve**. That is because, for calculating both precision and recall, we do not use the true negatives, and only consider the correct prediction of the minority class (the positive one).

We start by extracting the predicted scores/probabilities and calculating precision and recall for different thresholds:

```
y_pred_prob = tree_classifier.predict_proba(X_test_ohe)[:, 1]

precision, recall, _ = metrics.precision_recall_curve(y_test,
                                                       y_pred_prob)
```



As we do not actually need the thresholds, we substitute that output of the function with an underscore.

Having calculated the required elements, we can plot the curve:

```
ax = plt.subplot()
ax.plot(recall, precision,
         label=f"PR-AUC = {metrics.auc(recall, precision):.2f}")
ax.set(title="Precision-Recall Curve",
       xlabel="Recall",
       ylabel="Precision")
ax.legend()
```

Executing the snippet generates the following plot:

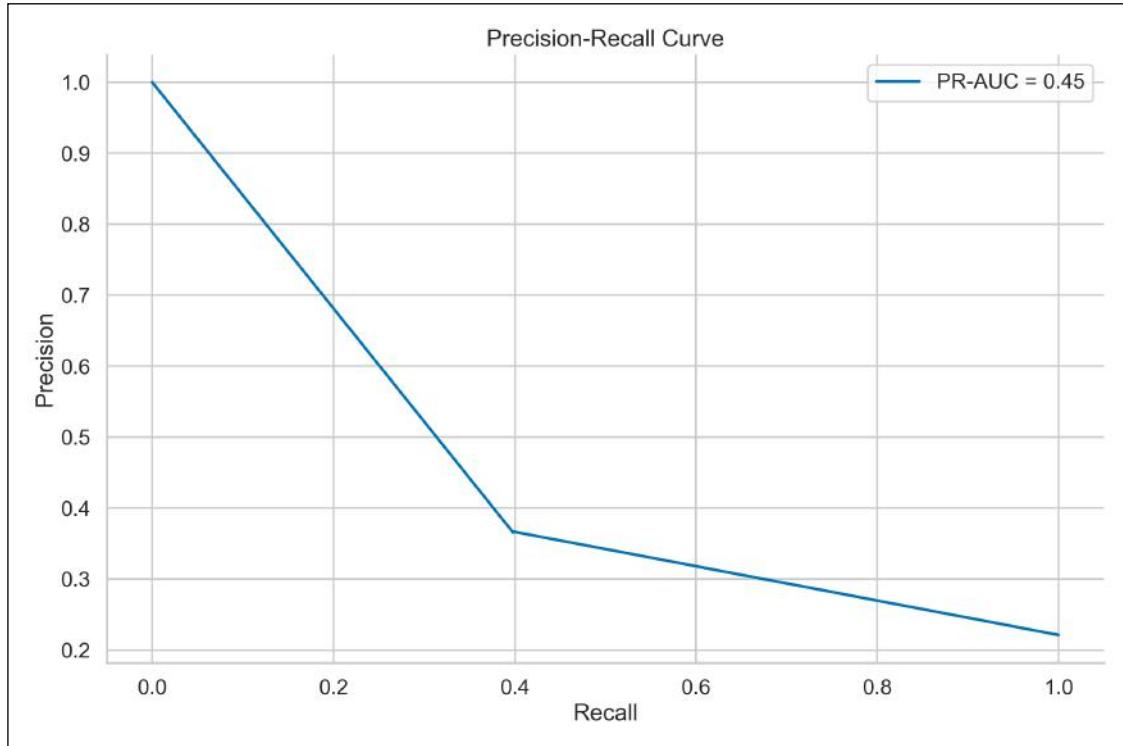


Figure 13.22: Precision-recall curve of the fitted decision tree classifier

Similar to the ROC curve, we can analyze the Precision-Recall curve as follows:

- Each point in the curve corresponds to the values of precision and recall for a different decision threshold.
- A decision threshold of 0 results in precision = 0 and recall = 1.
- A decision threshold of 1 results in precision = 1 and recall = 0.
- As a summary metric, we can approximate the area under the Precision-Recall curve.
- The PR-AUC ranges from 0 to 1, where 1 indicates the perfect model.

- A model with a PR-AUC of 1 can identify all the positive observations (perfect recall), while not wrongly labeling a single negative observation as a positive one (perfect precision). The perfect point is located in (1, 1), that is, the top-right corner of the plot.
- We can consider models that bow toward the (1, 1) point as skillful.

One potential issue with the PR-Curve in *Figure 13.22* is that it might be overly optimistic due to the undertaken interpolations when plotting the values of precision and recall for each threshold. A more realistic representation can be obtained using the following snippet:

```
ax = metrics.PrecisionRecallDisplay.from_estimator(  
    tree_classifier, X_test_ohe, y_test  
)  
ax.ax_.set_title("Precision-Recall Curve")
```

Executing the snippet generates the following plot:

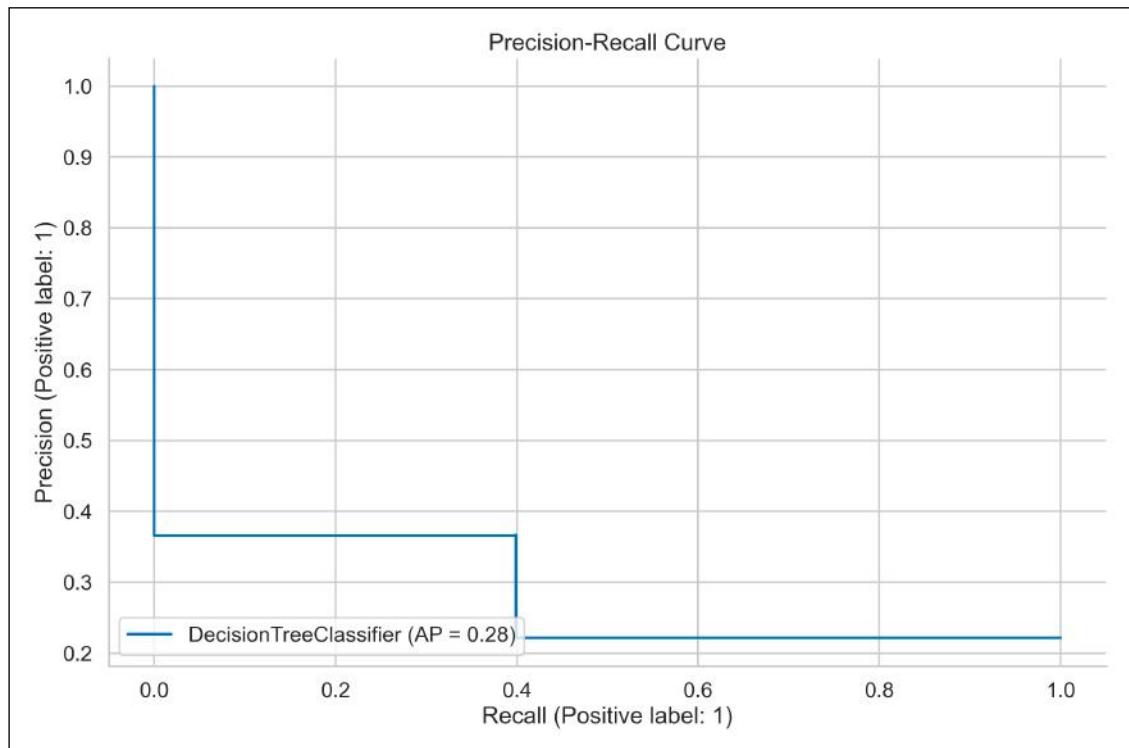


Figure 13.23: More realistic precision-recall curve of the fitted decision tree classifier

First, we can see that even though the shape is different, we can easily recognize the pattern and what the interpolation actually does. We can imagine connecting the extreme points of the plot with the single point (values of ~0.4 for both metrics), which would result in the shape obtained using interpolation.

Second, we can also see that the score decreased quite substantially (from 0.45 to 0.28). In the first case, we obtained the score using trapezoidal interpolation of the PR curve (`auc(precision, recall)` in `scikit-learn`). In the second case, the score is actually another metric—average precision. **Average precision** summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, where the weights are calculated as the increase in recall from the previous threshold.

Even though these two metrics produce very similar estimates in many cases, they are fundamentally different. The first approach uses an overly optimistic linear interpolation and its effect might be more pronounced when the data is highly skewed/imbalanced.

We have already covered the F1-Score, which was the harmonic mean of precision and recall. Actually, it is a specific case of a more general metric called the $F\beta$ -Score, where the β factor defines how much weight is put on recall, while precision has a weight of 1. To make sure that the weights sum up to one, both are normalized by dividing them by $(\beta + 1)$. Such a definition of the score implies the following:

- $\beta > 1$ —more weight is put on recall
- $\beta = 1$ —the same as the F1-Score, so recall and precision are treated equally
- $\beta < 1$ —more weight is put on precision

Some potential pitfalls of using precision, recall, or F1-Score include the fact that those metrics are asymmetric, that is, they focus on the positive class. When looking at their formulas, we can clearly see that they never account for the true negative category. That is exactly what **Matthew's correlation coefficient** (also known as the *phi-coefficient*) is trying to overcome:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}$$

Analyzing the formula reveals the following:

- All of the elements of the confusion matrix are taken into account while calculating the score
- The formula looks similar to the one used for calculating Pearson's correlation
- MCC treats the true class and the predicted class as two binary variables, and effectively calculates their correlation coefficient

MCC has values between -1 (a classifier always misclassifying) and 1 (a perfect classifier). A value of 0 indicates that the classifier is no better than random guessing. Overall, as MCC is a symmetric metric, in order to achieve a high value the classifier must be doing well in predicting both the positive and negative classes.



As MCC is not as intuitive and easy to interpret as F1-Score, it might be a good metric to use when the cost of low precision and low recall is unknown or unquantifiable. Then, MCC can be better than F1-Score as it provides a more balanced (symmetric) evaluation of a classifier.

Visualizing decision trees using dtreeviz

The default plotting functionalities in `scikit-learn` can definitely be considered good enough for visualizing decision trees. However, we can take it a step further using the `dtreeviz` library.

First, we import the library:

```
from dtreeviz.trees import *
```

Then, we train a smaller decision tree with a maximum depth of 3. We do so just in order to make the visualization easier to read. Unfortunately, there is no option in `dtreeviz` to plot only x levels of a tree:

```
small_tree = DecisionTreeClassifier(max_depth=3,
                                     random_state=42)
small_tree.fit(X_train_ohe, y_train)
```

Lastly, we plot the tree:

```
viz = dtreeviz(small_tree,
                x_data=X_train_ohe,
                y_data=y_train,
                feature_names=X_train_ohe.columns,
                target_name="Default",
                class_names=["No", "Yes"],
                title="Decision Tree - Loan default dataset")
viz
```

Running the snippet generates the following plot:

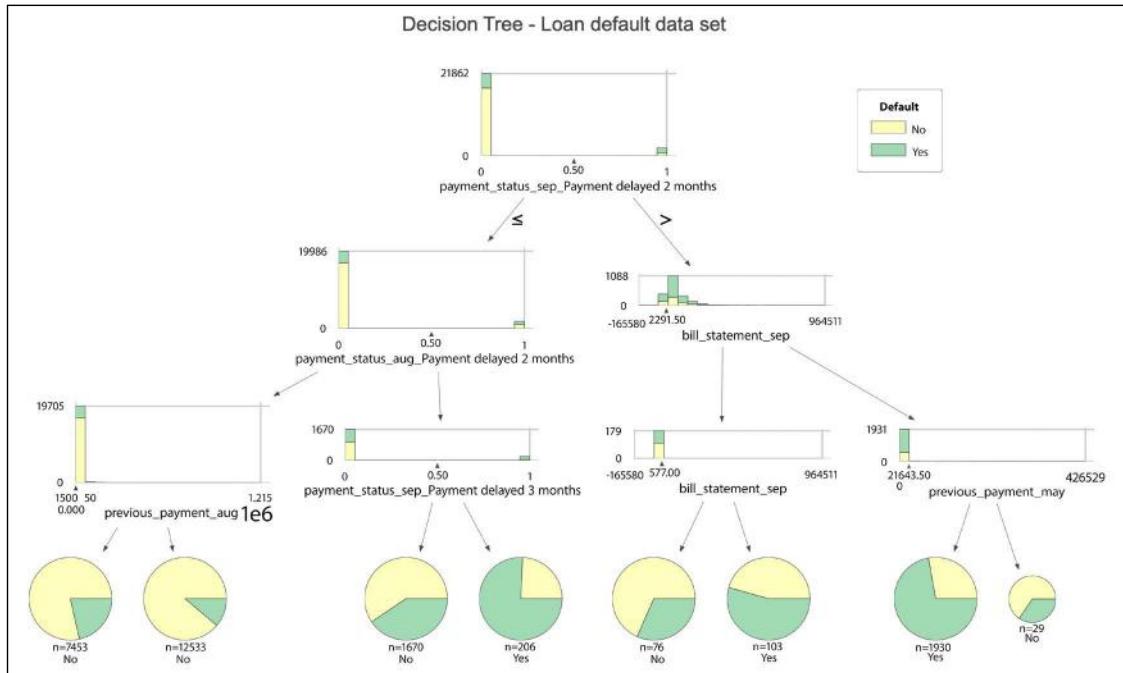


Figure 13.24: Decision tree visualized using dtreeviz

Compared to the previously generated plots, the ones created with `dtreeviz` additionally show the distribution of the feature used for splitting (separately for each class) together with the split value. What is more, the leaf nodes are presented as pie charts.

For more examples of using `dtreeviz`, including adding a path following a particular observation through all the splits in the tree, please refer to the notebook in the book's GitHub repository.

See also

Information on the dangers of using ROC-AUC as a performance evaluation metric:

- Lobo, J. M., Jiménez-Valverde, A., & Real, R. (2008). "AUC: a misleading measure of the performance of predictive distribution models." *Global Ecology and Biogeography*, 17(2), 145-151.
- Sokolova, M. & Lapalme, G. (2009). "A systematic analysis of performance measures for classification tasks." *Information Processing and Management*, 45(4), 427-437.

More information about the Precision-Recall curve:

- Davis, J. & Goadrich, M. (2006, June). “The relationship between Precision-Recall and ROC curves.” In *Proceedings of the 23rd international conference on Machine learning* (pp. 233-240).

Additional resources on decision trees:

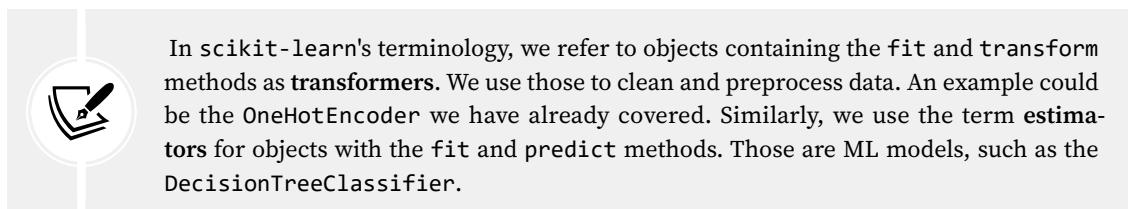
- Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984) *Classification and Regression Trees*. Chapman & Hall, Wadsworth, New York.
- Breiman, L. (2017). *Classification and Regression Trees*. Routledge.

Organizing the project with pipelines

In the previous recipes, we showed all the steps required to build a machine learning model—starting with loading data, splitting it into training and test sets, imputing missing values, encoding categorical features, and ultimately fitting a decision tree classifier.

The process requires multiple steps to be executed in a certain order, which can sometimes be tricky with a lot of modifications to the pipeline mid-work. That is why `scikit-learn` introduced pipelines. By using pipelines, we can sequentially apply a list of transformations to the data, and then train a given estimator (model).

One important point to be aware of is that the intermediate steps of the pipeline must have the `fit` and `transform` methods, while the final estimator only needs the `fit` method.



Using pipelines has several benefits:

- The flow is much easier to read and understand—the chain of operations to be executed on given columns is clear.
- Makes it easier to avoid data leakage, for example, when scaling the training set and then using cross-validation.
- The order of steps is enforced by the pipeline.
- Increased reproducibility.

In this recipe, we show how to create the entire project’s pipeline, from loading the data to training the classifier.

How to do it...

Execute the following steps to build the project's pipeline:

1. Import the libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from chapter_13_utils import performance_evaluation_report
```

2. Load the data, separate the target, and create the stratified train-test split:

```
df = pd.read_csv("../Datasets/credit_card_default.csv",
                 na_values="")

X = df.copy()
y = X.pop("default_payment_next_month")

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    stratify=y,
    random_state=42
)
```

3. Prepare lists of numerical/categorical features:

```
num_features = X_train.select_dtypes(include="number") \
    .columns \
    .to_list()
cat_features = X_train.select_dtypes(include="object") \
    .columns \
    .to_list()
```

4. Define the numerical pipeline:

```
num_pipeline = Pipeline(steps=[  
    ("imputer", SimpleImputer(strategy="median"))  
])
```

5. Define the categorical pipeline:

```
cat_list = [  
    list(X_train[col].dropna().unique()) for col in cat_features  
]  
  
cat_pipeline = Pipeline(steps=[  
    ("imputer", SimpleImputer(strategy="most_frequent")),  
    ("onehot", OneHotEncoder(categories=cat_list, sparse=False,  
                             handle_unknown="error",  
                             drop="first"))  
])
```

6. Define the ColumnTransformer object:

```
preprocessor = ColumnTransformer(  
    transformers=[  
        ("numerical", num_pipeline, num_features),  
        ("categorical", cat_pipeline, cat_features)  
    ],  
    remainder="drop"  
)
```

7. Define the full pipeline including the decision tree model:

```
dec_tree = DecisionTreeClassifier(random_state=42)  
  
tree_pipeline = Pipeline(steps=[  
    ("preprocessor", preprocessor),  
    ("classifier", dec_tree)  
])
```

8. Fit the pipeline to the data:

```
tree_pipeline.fit(X_train, y_train)
```

Executing the snippet generates the following preview of the pipeline:

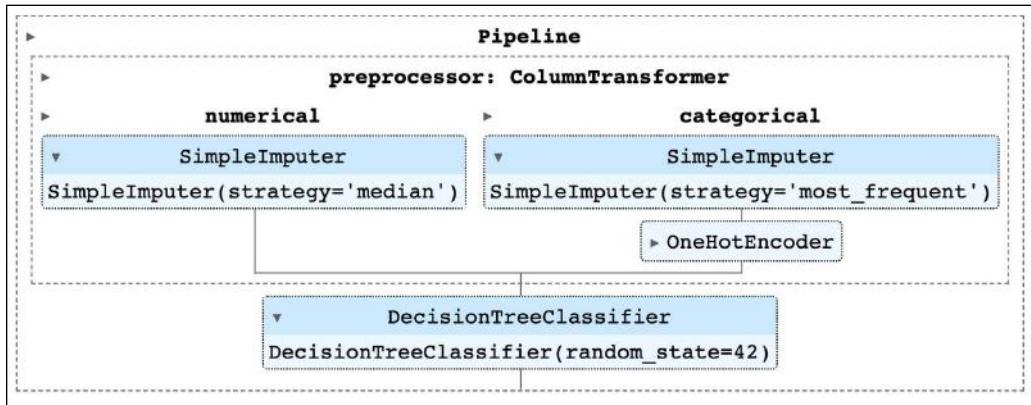


Figure 13.25: Preview of the pipeline

9. Evaluate the performance of the entire pipeline:

```

LABELS = ["No Default", "Default"]
tree_perf = performance_evaluation_report(tree_pipeline, X_test,
                                           y_test, labels=LABELS,
                                           show_plot=True)
  
```

Executing the snippet generates the following plot:

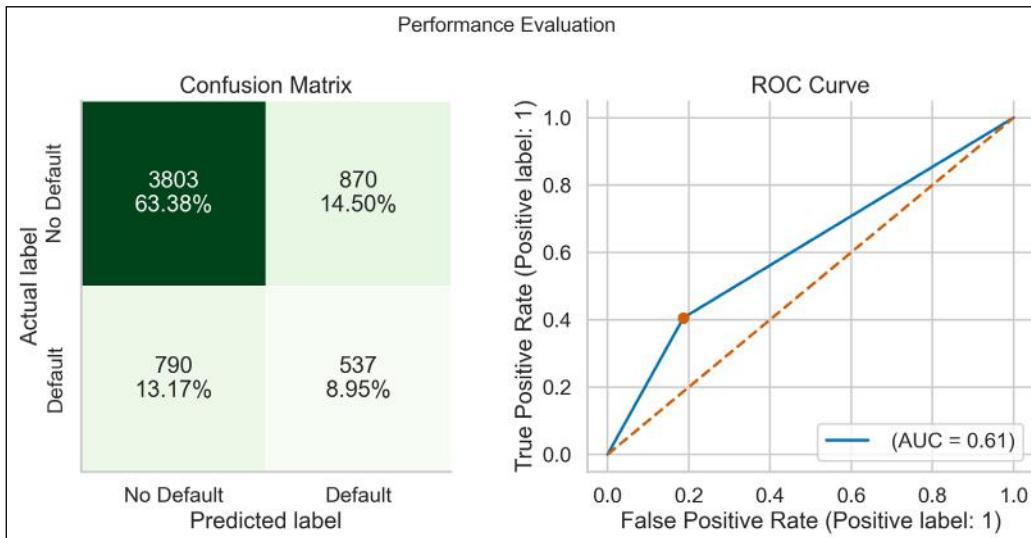


Figure 13.26: The performance evaluation report of the fitted pipeline

We see that the performance of the model is very similar to what we achieved by carrying out all the steps separately. Considering how little has changed, this is exactly what we expected to achieve.

How it works...

In *Step 1*, we imported the required libraries. The list can look a bit daunting, but that is due to the fact that we need to combine multiple functions/classes used in the previous recipes.

In *Step 2*, we loaded the data from a CSV file, separated the target variable from the features, and lastly created a stratified train-test split. Then, we also created two lists containing the names of the numerical and categorical features. We did so as we will apply different transformations depending on the data type of the feature. To select the appropriate columns, we used the `select_dtypes` method.

In *Step 4*, we defined the first `Pipeline` containing the transformations we wanted to apply to numerical features. As a matter of fact, we only wanted to impute the missing values of the features using the median value. While creating an instance of the `Pipeline` class, we provided a list of tuples containing the steps, each of the tuples consisting of the name of the step (for easier identification and accessing) and the class we wanted to use. In this case, it was the `SimpleImputer` class we covered in the *Identifying and dealing with missing values* recipe.

In *Step 5*, we prepared a similar pipeline for categorical features. This time, however, we chained two different operations—the imputer (using the most frequent value) and the one-hot encoder. For the encoder, we also specified a list of lists called `cat_list`, in which we listed all the possible categories. We based that information only on `X_train`. We did so as preparation for the next recipe, in which we introduce cross-validation, during which it can happen that some of the random draws will not contain all of the available categories.

In *Step 6*, we defined the `ColumnTransformer` object. In general, we use a `ColumnTransformer` when we want to apply separate transformations to different groups of columns/features. In our case, we have separate pipelines for numerical and categorical features. Again, we passed a list of tuples, where each tuple contains a name, one of the pipelines we defined before, and a list of columns to which the transformations should be applied. We also specified `remainder="drop"`, to drop any extra columns to which no transformations were applied. In this case, the transformations were applied to all features, so no columns were dropped. One thing to bear in mind is that `ColumnTransformer` returns numpy arrays instead of pandas DataFrames!



Another useful class available in `scikit-learn` is `FeatureUnion`. We can use it when we want to transform the same input data in different ways and then use those outputs as features. For example, we could be working with text data and want to apply two transformations: TF-IDF (term frequency-inverse document frequency) vectorization and extracting the text's length. The outputs of those should be appended to the original DataFrame, so we could use them as features for our model.

In *Step 7*, we once again used a `Pipeline` to chain the preprocessor (the previously defined `ColumnTransformer` object) with the decision tree classifier (for reproducibility's sake, we set the random state to 42). The last two steps involved fitting the entire pipeline to the data and using the custom function to evaluate its performance.



The `performance_evaluation_report` function was built in such a way that it works with any estimator or Pipeline that has the `predict` and `predict_proba` methods. Those are used to obtain predictions and their corresponding scores/probabilities.

There's more...

Adding custom transformers to a pipeline

In this recipe, we showed how to create the entire pipeline for a data science project. However, there are many other transformations we can apply to data as preprocessing steps. Some of them include:

- **Scaling numerical features:** In other words, changing the range of the features due to the fact that different features are measured on different scales, as that can introduce bias to the model. We should mostly be concerned with feature scaling when dealing with models that calculate some kind of distance between features (such as k-Nearest Neighbors) or linear models. Some popular scaling options from `scikit-learn` include `StandardScaler` and `MinMaxScaler`.
- **Discretizing continuous variables:** We can transform a continuous variable (such as age) into a finite number of bins (for example: <25, 26-50, and >51 years). When we want to create specific bins, we can use the `pd.cut` function, while `pd.qcut` can be used for splitting based on quantiles.
- **Transforming/removing outliers:** During the EDA, we often see feature values that are extreme and can be caused by some kind of error (for example, adding an extra digit to the age) or are simply incompatible with the rest (for example, a multimillionaire among a sample of middle-class citizens). Such outliers can skew the results of the model, and it is good practice to somehow deal with them. One solution would be to remove them altogether, but this can have an impact on the model's ability to generalize. We can also bring them closer to regular values.



ML models based on decision trees do not require any scaling.

In this example, we show how to create a custom transformer to detect and modify outliers. We apply a simple rule of thumb—we cap the values above/below the average ± 3 standard deviations. We create a dedicated transformer for this task, so we can incorporate the outlier treatment into the previously established pipeline:

1. Import the base estimator and transformer classes from `sklearn`:

```
from sklearn.base import BaseEstimator, TransformerMixin  
import numpy as np
```

In order for the custom transformer to be compatible with `scikit-learn`'s pipelines, it must have methods such as `fit`, `transform`, `fit_transform`, `get_params`, and `set_params`.

We could define all of those manually, but a definitely more appealing approach is to use Python's **class inheritance** to make the process easier. That is why we imported the `BaseEstimator` and `TransformerMixin` classes from `scikit-learn`. By inheriting from `TransformerMixin`, we do not need to specify the `fit_transform` method, while inheriting from `BaseEstimator` automatically provides the `get_params` and `set_params` methods.



As a learning experience, it definitely makes sense to dive into the code of at least some of the more popular transformers/estimators in `scikit-learn`. By doing so, we can learn a lot about the best practices of object-oriented programming and observe (and appreciate) how all of those classes consistently follow the same set of guidelines/principles.

2. Define the `OutlierRemover` class:

```
class OutlierRemover(BaseEstimator, TransformerMixin):
    def __init__(self, n_std=3):
        self.n_std = n_std

    def fit(self, X, y = None):
        if np.isnan(X).any(axis=None):
            raise ValueError("""Missing values in the array!
                            Please remove them.""")

        mean_vec = np.mean(X, axis=0)
        std_vec = np.std(X, axis=0)

        self.upper_band_ = pd.Series(
            mean_vec + self.n_std * std_vec
        )
        self.upper_band_ = (
            self.upper_band_.to_frame().transpose()
        )
        self.lower_band_ = pd.Series(
            mean_vec - self.n_std * std_vec
        )
        self.lower_band_ = (
            self.lower_band_.to_frame().transpose()
        )
        self.n_features_ = len(self.upper_band_.columns)

    return self
```

```

def transform(self, X, y = None):
    X_copy = pd.DataFrame(X.copy())

    upper_band = pd.concat(
        [self.upper_band_] * len(X_copy),
        ignore_index=True
    )
    lower_band = pd.concat(
        [self.lower_band_] * len(X_copy),
        ignore_index=True
    )

    X_copy[X_copy >= upper_band] = upper_band
    X_copy[X_copy <= lower_band] = lower_band

    return X_copy.values

```

The class can be broken down into the following components:

- In the `__init__` method, we stored the number of standard deviations that determines whether observations will be treated as outliers (the default is 3)
- In the `fit` method, we stored the upper and lower thresholds for being considered an outlier, as well as the number of features in general
- In the `transform` method, we capped all the values exceeding the thresholds determined in the `fit` method



Alternatively, we could have used the `clip` method of a pandas DataFrame to cap the extreme values.

One known limitation of this class is that it does not handle missing values. That is why we raise a `ValueError` when there are any missing values. Also, we use the `OutlierRemover` after the imputation in order to avoid that issue. We could, of course, account for the missing values in the transformer, however, this would make the code longer and less readable. We leave this as an exercise for the reader. Please refer to the definition of `SimpleImputer` in `scikit-learn` for an example of how to mask missing values while building transformers.

3. Add the `OutlierRemover` to the numerical pipeline:

```

num_pipeline = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median")),
    ("outliers", OutlierRemover())
])

```

4. Execute the rest of the pipeline to compare the results:

```
preprocessor = ColumnTransformer(  
    transformers=[  
        ("numerical", num_pipeline, num_features),  
        ("categorical", cat_pipeline, cat_features)  
    ],  
    remainder="drop"  
)  
  
dec_tree = DecisionTreeClassifier(random_state=42)  
  
tree_pipeline = Pipeline(steps=[("preprocessor", preprocessor),  
                               ("classifier", dec_tree)])  
  
tree_pipeline.fit(X_train, y_train)  
  
tree_perf = performance_evaluation_report(tree_pipeline, X_test,  
                                         y_test, labels=LABELS,  
                                         show_plot=True)
```

Executing the snippet generates the following plot:

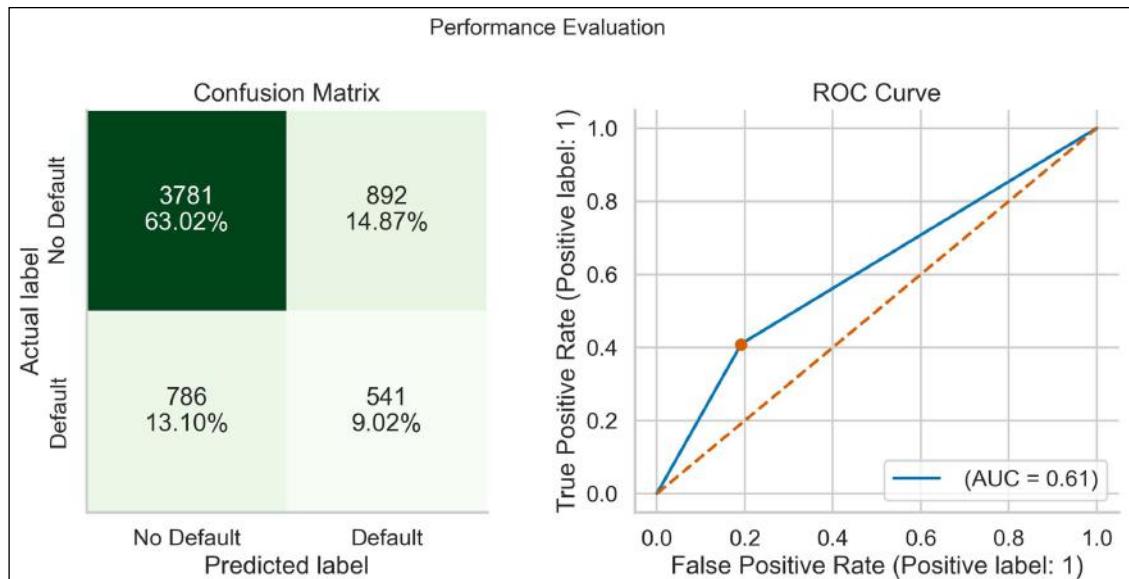


Figure 13.27: The performance evaluation report of the fitted pipeline (including treating outliers)

Including the outlier-capping transformation did not result in any significant changes in the performance of the entire pipeline.

Accessing the elements of the pipeline

While pipelines make our project easier to reproduce and less prone to data leakage, they come with a small disadvantage. Accessing the elements of a pipeline for further inspection or substitution becomes a bit more difficult. Let's illustrate with a few examples.

We start by displaying the entire pipeline represented as a dictionary by using the following snippet:

```
tree_pipeline.named_steps
```

Using that structure (not printed here for brevity), we can access the ML model at the end of the pipeline using the name we assigned to it:

```
tree_pipeline.named_steps["classifier"]
```

It gets a bit more complicated when we want to dive into the `ColumnTransformer`. Let's assume that we would like to inspect the upper bands (under the `upper_bands_` attribute) of the fitted `OutlierRemover`. To do so, we have to use the following snippet:

```
(  
    tree_pipeline  
    .named_steps["preprocessor"]  
    .named_transformers_["numerical"]["outliers"]  
    .upper_band_  
)
```

First, we followed the same approach as we have employed when accessing the estimator at the end of the pipeline. This time, we just used the name of the step containing the `ColumnTransformer`. Then, we used the `named_transformers_` attribute to access the deeper levels of the transformer. We selected the numerical pipeline and then the outlier treatment step using their corresponding names. Lastly, we accessed the upper bands of the custom transformer.



While accessing the steps of the `ColumnTransformer`, we could have used the `transformers_` attribute instead of the `named_transformers_`. However, then the output would be a list of tuples (the same ones as we have manually provided when defining the `ColumnTransformer`) and we have to access their elements using integer indices. We show how to access the upper bands using the `transformers_` attribute in the notebook available on GitHub.

Tuning hyperparameters using grid searches and cross-validation

In the previous recipes, we have used a decision tree model to try to predict whether a customer will default on their loan. As we have seen, the tree became quite massive with a depth of 44 levels, which prevented us from plotting it. However, this can also mean that the model is overfitted to the training data and will not perform well on unseen data. Maximum depth is actually one of the decision tree's hyperparameters, which we can tune to achieve better performance by finding a balance between underfitting and overfitting (bias-variance trade-off).

First, we outline some properties of hyperparameters:

- External characteristics of the model
- Not estimated based on data
- Can be considered the model's settings
- Set before the training phase
- Tuning them can result in better performance

We can also consider some properties of parameters:

- Internal characteristics of the model
- Estimated based on data, for example, the coefficients of linear regression
- Learned during the training phase

While tuning the model's hyperparameters, we would like to evaluate its performance on data that was not used for training. In the *Splitting data into training and test sets* recipe, we mentioned that we can create an additional validation set. The validation set is used explicitly to tune the model's hyperparameters, before the ultimate evaluation using the test set. However, creating the validation set comes at a price: data used for training (and possibly testing) is sacrificed, which can be especially harmful when dealing with small datasets.

That is the reason why **cross-validation** became so popular. It is a technique that allows us to obtain reliable estimates of the model's generalization error. It is easiest to understand how it works with an example. When doing k -fold cross-validation, we randomly split the training data into k folds. Then, we train the model using $k-1$ folds and evaluate the performance on the k^{th} fold. We repeat this process k times and average the resulting scores.

A potential drawback of cross-validation is the computational cost, especially when paired together with a grid search for hyperparameter tuning.



Figure 13.28: Scheme of a 5-fold cross-validation procedure

We already mentioned that **grid search** is a technique used for tuning hyperparameters. The underlying idea is to create a grid of all possible hyperparameter combinations and train the model using each one of them. Due to its exhaustive, brute-force search, the approach guarantees to find the optimal parameter within the grid. The drawback is that the size of the grid grows exponentially when adding more parameters or more considered values. The number of required model fits and predictions increases significantly if we additionally use cross-validation!

Let's illustrate this with an example by assuming that we are training a model with two hyperparameters: a and b . We define a grid that covers the following values of the hyperparameters: `{"a": [1, 2, 3], "b": [5, 6]}`. This means that there are 6 unique combinations of hyperparameters in our grid and the algorithm will fit the model 6 times. If we also use a 5-fold cross-validation procedure, it will result in 30 unique models being fitted in the grid search procedure!

As a potential solution to the problems encountered with grid search, we can also use the **random search** (also called **randomized grid search**). In this approach, we choose a random set of hyperparameters, train the model (also using cross-validation), return the scores, and repeat the entire process until we reach a predefined number of iterations or the computational time limit. Random search is preferred over grid search when dealing with a very large grid. That is because the former can explore a wider hyperparameter space and often find a hyperparameter set that performs very similarly to the optimal one (obtained from an exhaustive grid search) in a much shorter time. The only problematic question is: how many iterations are sufficient to find a good solution? Unfortunately, there is no simple answer to that. Most of the time, it is indicated by the available resources.

Getting ready

For this recipe, we use the decision tree pipeline created in the *Organizing the project with pipelines* recipe, including the outlier treatment from the *There's more...* section.

How to do it...

Execute the following steps to run both grid search and randomized search on the decision tree pipeline we have created in the *Organizing the project with pipelines* recipe:

1. Import the libraries:

```
from sklearn.model_selection import (
    GridSearchCV, cross_val_score,
    RandomizedSearchCV, cross_validate,
    StratifiedKFold
)
from sklearn import metrics
```

2. Define a cross-validation scheme:

```
k_fold = StratifiedKFold(5, shuffle=True, random_state=42)
```

3. Evaluate the pipeline using cross-validation:

```
cross_val_score(tree_pipeline, X_train, y_train, cv=k_fold)
```

Executing the snippet returns an array containing the estimator's default score (accuracy) values:

```
array([0.72333333, 0.72958333, 0.71375, 0.723125, 0.72])
```

4. Add extra metrics to the cross-validation:

```
cv_scores = cross_validate(
    tree_pipeline, X_train, y_train, cv=k_fold,
    scoring=["accuracy", "precision", "recall",
             "roc_auc"]
)
pd.DataFrame(cv_scores)
```

Executing the snippet generates the following table:

	fit_time	score_time	test_accuracy	test_precision	test_recall	test_roc_auc
0	0.800832	0.218017	0.723333	0.385604	0.424128	0.616333
1	0.746714	0.249949	0.729583	0.395575	0.420904	0.618938
2	0.759059	0.214318	0.713750	0.369783	0.417137	0.607940
3	0.764652	0.249994	0.723125	0.386383	0.427495	0.618066
4	0.717186	0.253843	0.720000	0.376748	0.405838	0.607129

Figure 13.29: The results of 5-fold cross-validation

In Figure 13.29, we can see the 4 requested metrics for each of the 5 cross-validation folds. The metrics have very similar values in each of the 5 test folds, which suggests that the cross-validation with stratified split worked as intended.

5. Define the parameter grid:

```
param_grid = {
    "classifier_criterion": ["entropy", "gini"],
    "classifier_max_depth": range(3, 11),
    "classifier_min_samples_leaf": range(2, 11),
    "preprocessor_numerical_outliers_n_std": [3, 4]
}
```

6. Run the exhaustive grid search:

```
classifier_gs = GridSearchCV(tree_pipeline, param_grid,
                             scoring="recall", cv=k_fold,
                             n_jobs=-1, verbose=1)

classifier_gs.fit(X_train, y_train)
```

Below we see how many models will be fitted using the exhaustive search:

```
Fitting 5 folds for each of 288 candidates, totalling 1440 fits
```

The best model from the exhaustive grid search is the following:

```
Best parameters: {'classifier_criterion': 'gini', 'classifier_max_depth': 10, 'classifier_min_samples_leaf': 7, 'preprocessor_numerical_outliers_n_std': 4}
Recall (Training set): 0.3858
Recall (Test set): 0.3775
```

7. Evaluate the performance of the tuned pipeline:

```
LABELS = ["No Default", "Default"]
tree_gs_perf = performance_evaluation_report(
    classifier_gs, X_test,
    y_test, labels=LABELS,
    show_plot=True
)
```

Executing the snippet generates the following plot:

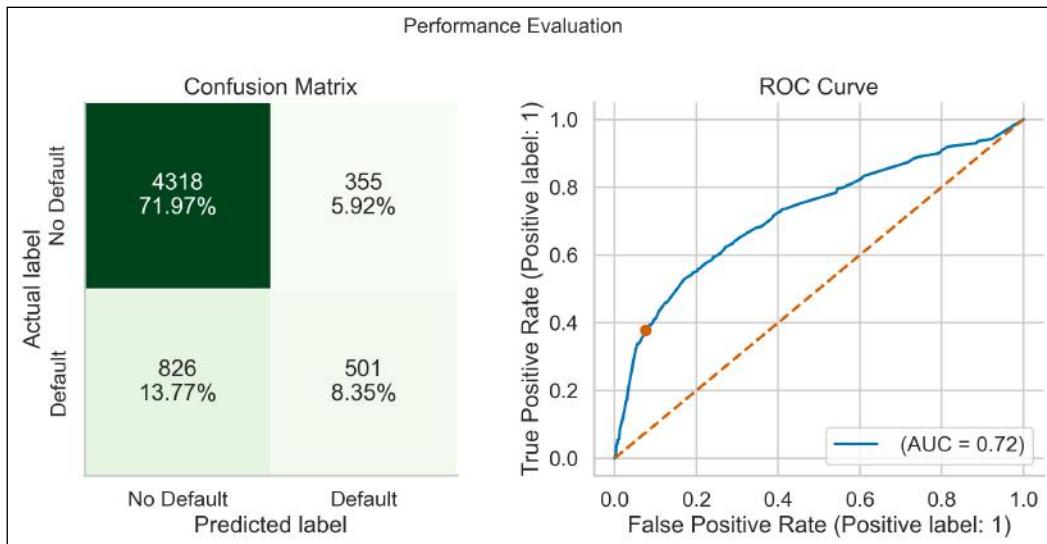


Figure 13.30: The performance evaluation report of the best pipeline identified by the exhaustive grid search

- Run the randomized grid search:

```
classifier_rs = RandomizedSearchCV(tree_pipeline, param_grid,
                                    scoring="recall", cv=k_fold,
                                    n_jobs=-1, verbose=1,
                                    n_iter=100, random_state=42)
classifier_rs.fit(X_train, y_train)

print(f"Best parameters: {classifier_rs.best_params_}")
print(f"Recall (Training set): {classifier_rs.best_score_:.4f}")
print(f"Recall (Test set): {metrics.recall_score(y_test, classifier_
rs.predict(X_test)):.4f}")
```

Below we can see that the randomized search will train fewer models than the exhaustive one:

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
```

The best model from the randomized grid search is the following:

```
Best parameters: {'preprocessor_numerical_outliers_n_std': 3,
'classifier_min_samples_leaf': 7, 'classifier_max_depth': 10,
'classifier_criterion': 'gini'}
Recall (Training set): 0.3854
Recall (Test set): 0.3760
```

In the randomized search, we looked at 100 random sets of hyperparameters, which correspond to ~1/3 of all possibilities covered by the exhaustive search. Even though the randomized search did not identify the same model as the best one, the performance of both pipelines is very similar on both training and test sets.

How it works...

In *Step 2*, we defined the 5-fold cross-validation scheme. As there is no inherent order in the data, we used shuffling and specified the random state for reproducibility. Stratification ensured that each fold received a similar ratio of classes in the target variable. Such a setting is crucial when we are dealing with imbalanced classes.

In *Step 3*, we evaluated the pipeline created in the *Organizing the project with pipelines* recipe using the `cross_val_score` function. We passed the estimator (the entire pipeline), the training data, and the cross-validation scheme as arguments to the function.

We could have also provided a number to the `cv` argument (the default is 5)—in the case of a classification problem, it would have automatically applied the stratified k -fold cross-validation. However, by providing a custom scheme, we also ensured that the random state was defined and that the results were reproducible.



We can clearly observe another advantage of using pipelines—we are not leaking any information while carrying out cross-validation. Without pipelines, we would fit our transformers (for example, `StandardScaler`) using the training data and then transform training and test sets separately. This way, we are not leaking any information from the test set. However, we are leaking a bit of information if we carry out cross-validation on such a transformed training set. That is because the folds used for validation were transformed using all the information from the training set.

In *Step 4*, we extended the cross-validation by using the `cross_validate` function. This function is more flexible in the way it allows us to use multiple evaluation criteria (we used accuracy, precision, recall, and the ROC AUC). Additionally, it records the time spent in both the training and inference steps. We printed the results in the form of a pandas DataFrame to make them easier to read. By default, the output of the function is a dictionary.

In *Step 5*, we defined the parameter grid to be used for the grid search. An important point to remember here is the naming convention when working with `Pipeline` objects. The keys in the grid dictionary are built from the name of the step/model concatenated with the hyperparameter name using a double underscore. In this example, we searched a space created on top of three hyperparameters of the decision tree classifier:

- `criterion`—the metric used for determining a split, either entropy or Gini importance.
- `max_depth`—the maximum depth of the tree.
- `min_samples_leaf`—the minimum number of observations in a leaf. It prevents the creation of trees with very few observations in leaves.

Additionally, we experimented with the outlier transformer, by using either three or four standard deviations from the mean to indicate whether an observation was an outlier. Please pay attention to the construction of the name, which contains the following pieces of information in sequence:

- `preprocessor`—the step of the pipeline.
- `numerical`—which pipeline it was within the `ColumnTransformer`.
- `outliers`—which step of that inner pipeline we are accessing.
- `n_std`—the name of the hyperparameter we wanted to specify.



When only tuning the estimator (model), we should directly use the names of the hyperparameters.

We decided to select the best-performing decision tree model based on recall, that is, the percentage of all defaults correctly identified by the model. This evaluation metric is definitely useful in cases when we are dealing with imbalanced classes, for example, when predicting default or fraud. In real life, there is often a different cost of a false negative (predicting no default when the user actually defaulted) and a false positive (predicting that a good customer defaults). To predict defaults, we decided that we could accept the cost of more false positives, in return for reducing the number of false negatives (missed defaults).

In *Step 6*, we created an instance of the `GridSearchCV` class. We provided the pipeline and parameter grid as inputs. We also specified recall as the scoring metric to be used for selecting the best model (different metrics could have been used here). We also used our custom CV scheme and indicated that we wanted to use all available cores to speed up the computations (`n_jobs=-1`).



When working with the grid search classes of `scikit-learn`, we can actually provide multiple evaluation metrics (specified as a list or dictionary). That is definitely helpful when we want to carry out a more in-depth analysis of the fitted models. We need to remember that when using multiple metrics, we must use the `refit` argument to specify which metric should be used to determine the best combination of hyperparameters.

We then used the `fit` method of the `GridSearchCV` object, just like any other estimator in `scikit-learn`. From the output, we saw that the grid contained 288 different combinations of hyperparameters. For each set, we fitted five models (5-fold cross-validation).



`GridSearchCV`'s default setting of `refit=True` means that after the entire grid search is completed, the best model has automatically been fitted once again, this time to the entire training set. We can then directly use that estimator (identified by the indicated criterion) for inference by running `classifier_gs.predict(X_test)`.

In Step 8, we created an instance of a randomized grid search. It is similar to a regular grid search, except that the maximum number of iterations was specified. In this case, we tested 100 different combinations from the parameter grid, which was roughly 1/3 of all available combinations.

There is one additional difference between the exhaustive and randomized approaches to grid search. In the latter one, we can provide a hyperparameter distribution instead of a list of distinct values. For example, let's assume that we have a hyperparameter that describes a ratio between 0 and 1. In the exhaustive grid search, we might specify the following values: [0, 0.2, 0.4, 0.6, 0.8, 1]. In the randomized search, we can use the same values and the search will randomly (uniformly) pick up a value from the list (there is no guarantee that all of them will be tested). Alternatively, we might prefer to draw a random value from the uniform distribution (restricted to values between 0 and 1) as the hyperparameter's value.



Under the hood, `scikit-learn` applies the following logic. If all hyperparameters are presented as lists, the algorithm performs sampling without replacement. If at least one hyperparameter is represented by a distribution, sampling with replacement is used instead.

There's more...

Faster search with successive halving

For each candidate set of hyperparameters, both exhaustive and random approaches to grid search train a model/pipeline using all available data. `scikit-learn` offers an additional approach to grid search called halving grid search, which is based on the idea of **successive halving**.

The algorithm works as follows. First, all candidate models are fitted using a small subset of the available training data (in general, using a limited amount of resources). Then, the best-performing candidates are picked out. In the next step, those best-performing candidates are retrained with a bigger subset of the training data. Those steps are repeated until the best set of hyperparameters is identified. In this approach, after each iteration, the number of available hyperparameter candidates is decreasing while the size of the training data (resources) is increasing.



The default behavior of the halving grid search is to use training data as a resource. However, we could just as well use another hyperparameter of the estimator we are trying to tune, as long as it accepts positive integer values. For example, we could use the number of trees (`n_estimators`) of the Random Forest model as the resource to be increased with each iteration.

The speed of the algorithm depends on two hyperparameters:

- `min_resources`—the minimum amount of resources that any candidate is allowed to use. In practice, this corresponds to the number of resources used in the first iteration.

- `factor`—the halving parameter. The reciprocal of the `factor` (`1 / factor`) determines the proportion of candidates to be selected as the best models in each iteration. The product of the `factor` and the previous iteration's number of resources determines the current iteration's number of resources.

While picking those two might seem a bit daunting with all the calculations to be carried out manually to make use of most of the resources, `scikit-learn` makes it easier for us with the "exhaust" value of the `min_resources` argument. Then, the algorithm will determine for us the number of the resources in the first iteration such that the last iteration uses as many resources as possible. In the default case, it will result in the last iteration using as much of the training data as possible.



Similar to the randomized grid search, `scikit-learn` also offers a randomized halving grid search. The only difference compared to what we have already described is that at the very beginning, a fixed number of candidates is sampled at random from the parameter space. This number is determined by the `n_candidates` argument.

Below we demonstrate how to use the `HalvingGridSearchCV`. First, we need to explicitly allow using the experimental feature before importing it (in the future, this step might be redundant when the feature is no longer experimental):

```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingGridSearchCV
```

Then, we find the best hyperparameters for our decision tree pipeline:

```
classifier_sh = HalvingGridSearchCV(tree_pipeline, param_grid,
                                      scoring="recall", cv=k_fold,
                                      n_jobs=-1, verbose=1,
                                      min_resources="exhaust", factor=3)

classifier_sh.fit(X_train, y_train)
```

We can see how the successive halving algorithm works in practice in the following log:

```
n_iterations: 6
n_required_iterations: 6
n_possible_iterations: 6
min_resources_: 98
max_resources_: 24000
aggressive_elimination: False
factor: 3
-----
iter: 0
n_candidates: 288
```

```

n_resources: 98
Fitting 5 folds for each of 288 candidates, totalling 1440 fits
-----
iter: 1
n_candidates: 96
n_resources: 294
Fitting 5 folds for each of 96 candidates, totalling 480 fits
-----
iter: 2
n_candidates: 32
n_resources: 882
Fitting 5 folds for each of 32 candidates, totalling 160 fits
-----
iter: 3
n_candidates: 11
n_resources: 2646
Fitting 5 folds for each of 11 candidates, totalling 55 fits
-----
iter: 4
n_candidates: 4
n_resources: 7938
Fitting 5 folds for each of 4 candidates, totalling 20 fits
-----
iter: 5
n_candidates: 2
n_resources: 23814
Fitting 5 folds for each of 2 candidates, totalling 10 fits

```

As we have mentioned before, `max_resources` is determined by the size of the training data, that is, 24,000 observations. Then, the algorithm figured out that it needs to start with a sample size of 98 in order to end the procedure with as big a sample as possible. In this case, in the last iteration, the algorithm used 23,814 training observations.

In the following table, we can see which values of the hyperparameters were picked by each of the 3 approaches to grid search we have covered in this recipe. They are very similar, and so is their performance on the test set (the exact comparison is available in the notebook on GitHub). We leave the comparison of fitting times of all those algorithms as an exercise for the reader.

	<code>classifier__criterion</code>	<code>classifier__max_depth</code>	<code>classifier__min_samples_leaf</code>	<code>preprocessor__numerical_outliers__n_std</code>
<code>grid_search</code>	gini	10	7	4
<code>randomized_search</code>	gini	10	7	3
<code>halving_search</code>	gini	10	6	4

Figure 13.31: The best values of hyperparameters identified by exhaustive, randomized, and halving grid search

Grid search with multiple classifiers

We can also create a grid containing multiple classifiers. This way, we can see which model performs best with our data. To do so, we first import another classifier from `scikit-learn`. We will use the famous Random Forest:

```
from sklearn.ensemble import RandomForestClassifier
```

We selected this model as it is an ensemble of decision trees and thus also does not require any further preprocessing of the data. For example, if we wanted to use a simple logistic regression classifier (with regularization), we should also scale the features (standardize/normalize) by adding an additional step to the numerical part of the preprocessing pipeline. We cover the Random Forest model in more detail in the next chapter.

Again, we need to define the parameter grid. This time, it is a list containing multiple dictionaries—one dictionary per classifier. The hyperparameters for the decision tree are the same as before, and we chose the simplest hyperparameters of the Random Forest, as those do not require additional explanations.

It is worth mentioning that if we want to tune some other hyperparameters in the pipeline, we need to specify them in each of the dictionaries in the list. That is why `preprocessor_numerical_outliers_n_std` is included twice in the following snippet:

```
param_grid = [
    {"classifier": [RandomForestClassifier(random_state=42)],
     "classifier_n_estimators": np.linspace(100, 500, 10, dtype=int),
     "classifier_max_depth": range(3, 11),
     "preprocessor_numerical_outliers_n_std": [3, 4]},
    {"classifier": [DecisionTreeClassifier(random_state=42)],
     "classifier_criterion": ["entropy", "gini"],
     "classifier_max_depth": range(3, 11),
     "classifier_min_samples_leaf": range(2, 11),
     "preprocessor_numerical_outliers_n_std": [3, 4]}
]
```

The rest of the process is exactly the same as before:

```
classifier_gs_2 = GridSearchCV(tree_pipeline, param_grid,
                                scoring="recall", cv=k_fold,
                                n_jobs=-1, verbose=1)

classifier_gs_2.fit(X_train, y_train)

print(f"Best parameters: {classifier_gs_2.best_params_}")
print(f"Recall (Training set): {classifier_gs_2.best_score_:.4f}")
print(f"Recall (Test set): {metrics.recall_score(y_test, classifier_gs_2.
predict(X_test)):.4f}")
```

Running the snippet generates the following output:

```
Best parameters: {'classifier': DecisionTreeClassifier(max_depth=10,
min_samples_leaf=7, random_state=42), 'classifier_criterion': 'gini',
'classifier_max_depth': 10, 'classifier_min_samples_leaf': 7, 'preprocessor_
numerical_outliers_n_std': 4}
Recall (Training set): 0.3858
Recall (Test set): 0.3775
```

Turns out that the tuned decision tree managed to outperform an ensemble of trees. As we will see in the next chapter, we can easily change the outcome with a bit more tuning of the Random Forest classifier. After all, we have only tuned two of the many hyperparameters available.

We can use the following snippet to extract and print all the considered hyperparameter/classifier combinations, starting with the best one:

```
pd.DataFrame(classifier_gs_2.cv_results_).sort_values("rank_test_score")
```

See also

An additional resource on the randomized search procedure is available here:

- Bergstra, J. & Bengio, Y. (2012). “Random search for hyper-parameter optimization.” *Journal of Machine Learning Research*, 13(Feb), 281-305. <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.

Summary

In this chapter, we have covered the basics required to approach any machine learning project, not just limited to the financial domain. We did the following:

- Imported the data and optimized its memory usage
- Thoroughly explored the data (distributions of features, missing values, and class imbalance), which should already provide some ideas about potential feature engineering
- Identified the missing values in our dataset and imputed them
- Learned how to encode categorical variables so that they are correctly interpreted by machine learning models
- Fitted a decision tree classifier using the most popular and mature ML library—scikit-learn
- Learned how to organize our entire codebase using pipelines
- Learned how to tune the hyperparameters of the model to squeeze out some extra performance and find a balance between underfitting and overfitting

It is crucial to understand those steps and their significance, as they can be applied to any data science project, not only binary classification. The steps would be virtually the same for a regression problem, for example, predicting the price of a house. We would use slightly different estimators (though most of them work for both classification and regression) and evaluate the performance using different metrics (MSE, RMSE, MAE, MAPE, and so on). But the principles stay the same.

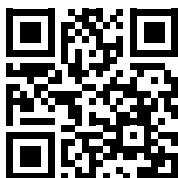
If you are interested in putting the knowledge from this chapter into practice, we can recommend the following sources for finding data for your next project:

- Google Datasets: <https://datasetsearch.research.google.com/>
- Kaggle: <https://www.kaggle.com/datasets>
- UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/index.php>

In the next chapter, we cover a selection of techniques that might be helpful in further improving the initial model. We will cover, among others, more complex classifiers, Bayesian hyperparameter tuning, dealing with class imbalance, exploring feature importance and selection, and more.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>

14

Advanced Concepts for Machine Learning Projects

In the previous chapter, we introduced a possible workflow for solving a real-life problem using machine learning. We went over the entire project, starting with cleaning the data, through training and tuning a model, and then lastly evaluating its performance. However, this is rarely the end of the project. In that project, we used a simple decision tree classifier, which most of the time can be used as a benchmark or minimum viable product (MVP). In this chapter, we cover a few more advanced concepts that can help with improving the value of the project and make it easier to adopt by the business stakeholders.

After creating the MVP, which serves as a baseline, we would like to improve the model's performance. While attempting to improve the model, we should also try to balance underfitting and overfitting. There are a few ways to do so, some of which include:

- Gathering more data (observations)
- Adding more features—either by gathering additional data (for example, by using external data sources) or through feature engineering using currently available information
- Using more complex models
- Selecting only the relevant features
- Tuning the hyperparameters

There is a common stereotype that data scientists spend 80% of their time on a project gathering and cleaning data while only 20% remains for the actual modeling. In line with the stereotype, adding more data might greatly improve a model's performance, especially when dealing with imbalanced classes in a classification problem. But finding additional data (be it observations or features) is not always possible, or might simply be too complicated. Then, the other solution may be to use more complex models or to tune the hyperparameters to squeeze out some extra performance.

We start the chapter by presenting how to use more advanced classifiers, which are also based on decision trees. Some of them (XGBoost and LightGBM) are frequently used for winning machine learning competitions (such as those found on Kaggle). Additionally, we introduce the concept of stacking multiple machine learning models to further improve prediction performance.

Another common real-life problem concerns dealing with imbalanced data, that is, when one class (such as default or fraud) is rarely observed in practice. This makes it especially difficult to train a model to accurately capture the minority class observations. We introduce a few common approaches to handling class imbalance and compare their performance on a credit card fraud dataset, in which the minority class corresponds to 0.17% of all the observations.

Then, we also expand on hyperparameter tuning, which was explained in the previous chapter. Previously, we used either an exhaustive grid search or a randomized search, both of which are carried out in an uninformed manner. This means that there is no underlying logic in selecting the next set of hyperparameters to investigate. This time, we introduce Bayesian optimization, in which past attempts are used to select the next set of values to explore. This approach can significantly speed up the tuning phase of our projects.

In many industries (and finance especially) it is crucial to understand the logic behind a model's prediction. For example, a bank might be legally obliged to provide actual reasons for declining a credit request, or it can try to limit its losses by predicting which customers are likely to default on a loan. To get a better understanding of the models, we explore various approaches to determining feature importance and model explainability. The latter is especially relevant when dealing with complex models, which are often considered to be black boxes, that is, unexplainable. We can additionally use those insights to select only the most relevant features, which can further improve the model's performance.

In this chapter, we present the following recipes:

- Exploring ensemble classifiers
- Exploring alternative approaches to encoding categorical features
- Investigating different approaches to handling imbalanced data
- Leveraging the wisdom of the crowds with stacked ensembles
- Bayesian hyperparameter optimization
- Investigating feature importance
- Exploring feature selection techniques
- Exploring explainable AI techniques

Exploring ensemble classifiers

In *Chapter 13, Applied Machine Learning: Identifying Credit Default*, we learned how to build an entire machine learning pipeline, which contained both preprocessing steps (imputing missing values, encoding categorical features, and so on) and a machine learning model. Our task was to predict customer default, that is, their inability to repay their debts. We used a decision tree model as the classifier.

Decision trees are considered simple models and one of their drawbacks is overfitting to the training data. They belong to the group of high-variance models, which means that a small change to the training data can greatly impact the tree's structure and its predictions. To overcome those issues, they can be used as building blocks for more complex models. **Ensemble models** combine predictions of multiple base models (for example, decision trees) in order to improve the final model's generalizability and robustness. This way, they transform the initial high-variance estimators into a low-variance aggregate estimator.

On a high level, we could divide the ensemble models into two groups:

- **Averaging methods**—several models are estimated independently and then their predictions are averaged. The underlying principle is that the combined model is better than a single one as its variance is reduced. Examples: Random Forest and Extremely Randomized Trees.
- **Boosting methods**—in this approach, multiple base estimators are built sequentially and each one tries to reduce the bias of the combined estimator. Again, the underlying assumption is that a combination of multiple weak models produces a powerful ensemble. Examples: Gradient Boosted Trees, XGBoost, LightGBM, and CatBoost.

In this recipe, we use a selection of ensemble models to try to improve the performance of the decision tree approach. As those models are based on decision trees, the same principles about feature scaling (no explicit need for it) apply and we can reuse most of the previously created pipeline.

Getting ready

In this recipe, we build on top of what we already established in the *Organizing the project with pipelines* recipe from the previous chapter, in which we created the default prediction pipeline, from loading the data to training the classifier.

In this recipe, we use the variant without the outlier removal procedure. We will be replacing the last step (the classifier) with more complex ensemble models. Additionally, we first fit the decision tree pipeline to the data to obtain the baseline model for performance comparison. For your convenience, we reiterate all the required steps in the notebook accompanying this chapter.

How to do it...

Execute the following steps to train the ensemble classifiers:

1. Import the libraries:

```
from sklearn.ensemble import (RandomForestClassifier,  
                               GradientBoostingClassifier)  
from xgboost.sklearn import XGBClassifier  
from lightgbm import LGBMClassifier  
from chapter_14_utils import performance_evaluation_report
```

In this chapter, we also use the already familiar `performance_evaluation_report` helper function.

2. Define and fit the Random Forest pipeline:

The performance of the Random Forest can be summarized by the following plot:

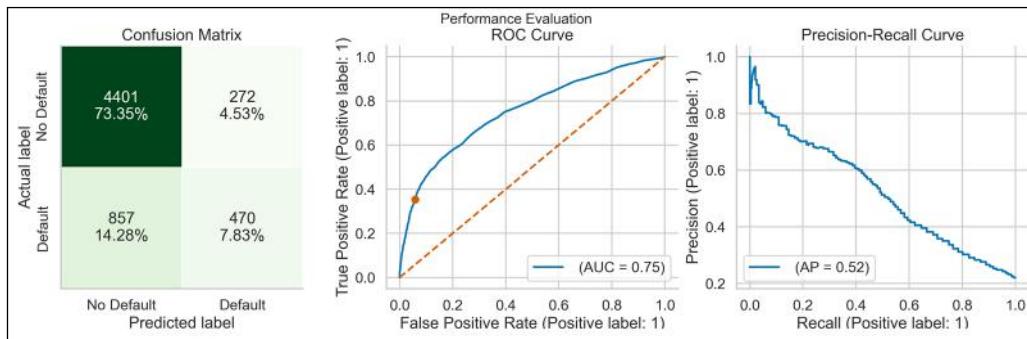


Figure 14.1: Performance evaluation of the Random Forest model

3. Define and fit the Gradient Boosted Trees pipeline:

The performance of the Gradient Boosted Trees can be summarized by the following plot:

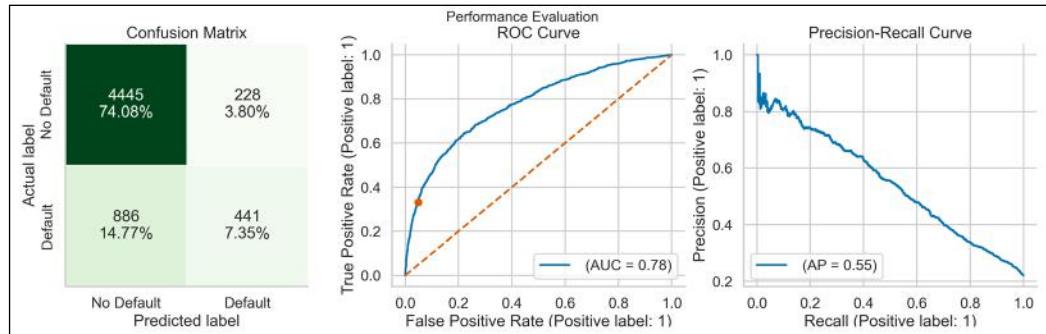


Figure 14.2: Performance evaluation of the Gradient Boosted Trees model

- Define and fit an XGBoost pipeline:

```
xgb = XGBClassifier(random_state=42)
xgb_pipeline = Pipeline(
    steps=[("preprocessor", preprocessor),
          ("classifier", xgb)])
)

xgb_pipeline.fit(X_train, y_train)
xgb_perf = performance_evaluation_report(xgb_pipeline, X_test,
                                         y_test, labels=LABELS,
                                         show_plot=True,
                                         show_pr_curve=True)
```

The performance of the XGBoost can be summarized by the following plot:

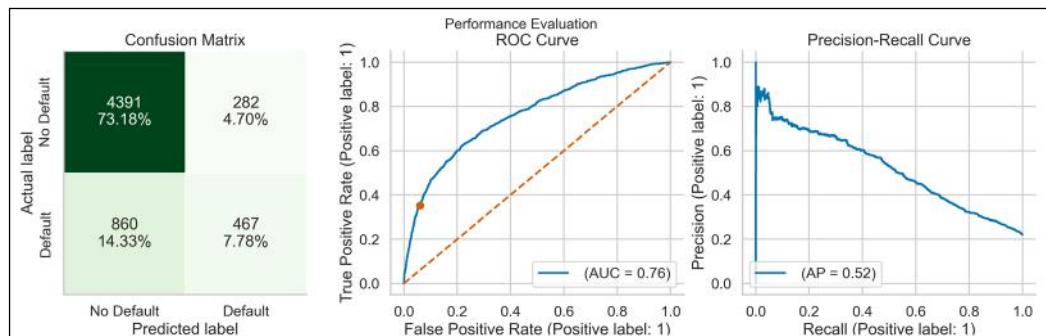


Figure 14.3: Performance evaluation of the XGBoost model

5. Define and fit the LightGBM pipeline:

```

lgbm = LGBMClassifier(random_state=42)
lgbm_pipeline = Pipeline(
    steps=[("preprocessor", preprocessor),
          ("classifier", lgbm)]
)

lgbm_pipeline.fit(X_train, y_train)
lgbm_perf = performance_evaluation_report(lgbm_pipeline, X_test,
                                           y_test, labels=LABELS,
                                           show_plot=True,
                                           show_pr_curve=True)

```

The performance of the LightGBM can be summarized by the following plot:

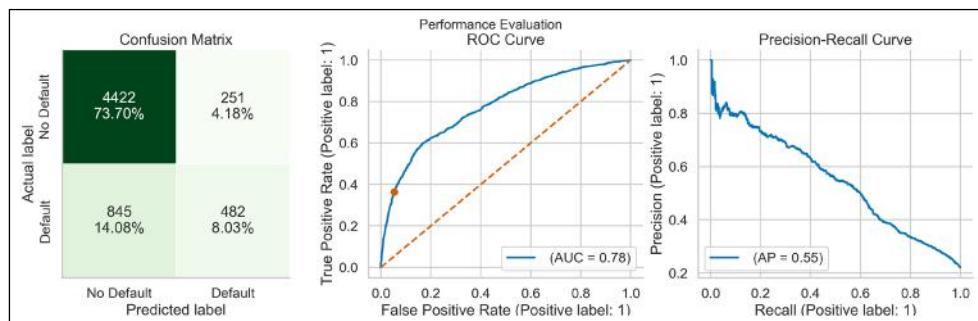


Figure 14.4: Performance evaluation of the LightGBM model

From the reports, it looks like the shapes of the ROC curve and the Precision-Recall curve were very similar for all the considered models. We will look at the scores of the models in the *There's more...* section.

How it works...

This recipe shows how easy it is to use different classifiers, as long as we want to use their default settings. In the first step, we imported the classifiers from their respective libraries.



In this recipe, we have used the `scikit-learn` API of libraries such as XGBoost or LightGBM. However, we could also use their native approaches to training models, which might require some additional effort, such as converting a `pandas DataFrame` to formats acceptable by those libraries. Using the native approaches can yield some extra benefits, for example, in terms of accessing certain hyperparameters or configuration settings.

In Steps 2 to 5, we created a separate pipeline for each classifier. We combined the already established `ColumnTransformer` preprocessor with the corresponding classifier. Then, we fitted each pipeline to the training data and presented the performance evaluation report.



Some of the considered ensemble models offer additional functionalities in the `fit` method (as opposed to setting hyperparameters when instantiating the class). For example, when using the `fit` method of LightGBM we can pass in the names/indices of categorical features. By doing so, the algorithm knows how to treat those features using its own approach, without the need for explicit one-hot encoding. Similarly, we could use a wide variety of available callbacks.

Thanks to modern Python libraries, fitting all the considered classifiers was extremely easy. We only had to replace the model's class in the pipeline with another one. Keeping in mind how simple it is to experiment with different models, it is good to have at least a basic understanding of what those models do and what their strengths and weaknesses are. That is why below we provide a brief introduction to the considered algorithms.

Random Forest

Random Forest is an example of an ensemble of models, that is, it trains multiple models (decision trees) and uses them to create predictions. In the case of a regression problem, it takes the average value of all the underlying trees. For classification it uses a majority vote. Random Forest offers more than just training many trees and aggregating their results.

First, it uses **bagging** (bootstrap aggregation)—each tree is trained on a subset of all available observations. Those are drawn randomly with replacement, so—unless specified otherwise—the total number of observations used for each tree is the same as the total in the training set. Even though a single tree might have high variance with respect to a particular dataset (due to bagging), the forest will have lower variance overall, without increasing the bias. Additionally, this approach can also reduce the effect of any outliers in the data as they will not be used in all of the trees. To add even more randomness, each tree only considers a subset of all features to create each split. We can control that number using a dedicated hyperparameter.

Thanks to those two mechanisms, the trees in the forest are not correlated with each other and are built independently. The latter allows for the parallelization of the tree-building step.

Random Forest provides a good trade-off between complexity and performance. Often—with any tuning—we can get much better performance than when using simpler algorithms, such as decision trees or linear/logistic regression. That is because Random Forest has a lower bias (due to its flexibility) and reduced variance (due to aggregating predictions of multiple models).

Gradient Boosted Trees

Gradient Boosted Trees is another type of ensemble model. The idea is to train many weak learners (shallow decision trees/stumps with high bias) and combine them to obtain a strong learner. In contrast to Random Forest, Gradient Boosted Trees is a sequential/iterative algorithm. In **boosting**, we start with the first weak learner, and each of the subsequent learners tries to learn from the mistakes of the previous ones. They do this by being fitted to the residuals (error terms) of the previous models.

The reason why we create an ensemble of weak learners instead of strong learners is that in the case of the strong learners, the errors/mislabeled data points would most likely be the noise in the data, so the overall model would end up overfitting to the training data.

The term *gradient* comes from the fact that the trees are built using **gradient descent**, which is an optimization algorithm. Without going into too much detail, it uses the gradient (slope) of the loss function to minimize the overall loss and achieve the best performance. The loss function represents the difference between the actual and predicted values. In practice, to perform the gradient descent procedure in Gradient Boosted Trees, we add such a tree to the model that follows the gradient. In other words, such a tree reduces the value of the loss function.

We can describe the boosting procedure using the following steps:

1. The process starts with a simple estimate (mean, median, and so on).
2. A tree is fitted to the error of that prediction.
3. The prediction is adjusted using the tree's prediction. However, it is not fully adjusted, but only to a certain degree (based on a learning rate hyperparameter).
4. Another tree is fitted to the error of the updated prediction and the prediction is further adjusted as in the previous step.
5. The algorithm continues to iteratively reduce the error until a specified number of rounds (or another stopping criterion) is reached.
6. The final prediction is the sum of the initial prediction and all the adjustments (predictions of the error weighted with the learning rate).



In contrast to Random Forest, Gradient Boosted Trees use all available data to train the models. However, we can use random sampling without replacement for each tree by using the **subsample** hyperparameter. Then, we are dealing with **Stochastic Gradient Boosted Trees**. Additionally, similarly to Random Forest, we can make the trees consider only a subset of features when making a split.

XGBoost

Extreme Gradient Boosting (XGBoost) is an implementation of Gradient Boosted Trees that incorporates a series of improvements resulting in superior performance (both in terms of evaluation metrics and estimation time). Since being published, the algorithm has been successfully used to win many data science competitions.

In this recipe, we only present a high-level overview of its distinguishable features. For a more detailed overview, please refer to the original paper (Chen *et al.* (2016)) or documentation. The key concepts of XGBoost are the following:

- XGBoost combines a pre-sorted algorithm with a histogram-based algorithm to calculate the best splits. This tackles a significant inefficiency of Gradient Boosted Trees, namely that the algorithm considers the potential loss for all possible splits when creating a new branch (especially important when considering hundreds or thousands of features).

- The algorithm uses the Newton-Raphson method to approximate the loss function, which allows us to use a wider variety of loss functions.
- XGBoost has an extra randomization parameter to reduce the correlation between the trees.
- XGBoost combines Lasso (L1) and Ridge (L2) regularization to prevent overfitting.
- It offers a more efficient approach to tree pruning.
- XGBoost has a feature called monotonic constraints—the algorithm sacrifices some accuracy and increases the training time to improve model interpretability.
- XGBoost does not take categorical features as input—we must use some kind of encoding for them.
- The algorithm can handle missing values in the data.

LightGBM

LightGBM, released by Microsoft, is another competition-winning implementation of Gradient Boosted Trees. Thanks to some improvements, LightGBM results in a similar performance to XGBoost, but with faster training time. Key features include the following:

- The difference in speed is caused by the approach to growing trees. In general, algorithms (such as XGBoost) use a level-wise (horizontal) approach. LightGBM, on the other hand, grows trees leaf-wise (vertically). The leaf-wise algorithm chooses the leaf with the maximum reduction in the loss function. Such algorithms tend to converge faster than the level-wise ones; however, they tend to be more prone to overfitting (especially with small datasets).
- LightGBM employs a technique called **Gradient-based One-Side Sampling (GOSS)** to filter out the data instances used for finding the best split value. Intuitively, observations with small gradients are already well trained, while those with large gradients have more room for improvement. GOSS retains instances with large gradients and additionally samples randomly from observations with small gradients.
- LightGBM uses **Exclusive Feature Bundling (EFB)** to take advantage of sparse datasets and bundles together features that are mutually exclusive (they never have values of zero at the same time). This leads to a reduction in the complexity (dimensionality) of the feature space.
- The algorithm uses histogram-based methods to bucket continuous feature values into discrete bins in order to speed up training and reduce memory usage.



The leaf-wise algorithm was later added to XGBoost as well. To make use of it, we need to set `grow_policy` to "lossguide".

There's more...

In this recipe, we showed how to use selected ensemble classifiers to try to improve our ability to predict customers' likelihood of defaulting their loan. To make things even more interesting, these models have dozens of hyperparameters to tune, which can significantly increase (or decrease) their performance.

For brevity, we will not discuss the hyperparameter tuning of these models here. We refer you to the accompanying Jupyter notebook for a short introduction to tuning these models using a randomized grid search approach. Here, we only present a table containing the results. We can compare the performance of the models with default settings versus their tuned counterparts.

	accuracy	precision	recall	specificity	f1_score	cohens_kappa	matthews_corr_coeff	roc_auc	pr_auc	average_precision
decision_tree_baseline	0.7233	0.3817	0.4047	0.8138	0.3928	0.2139	0.2140	0.6095	0.4589	0.2862
random_forest	0.8118	0.6334	0.3542	0.9418	0.4543	0.3514	0.3731	0.7518	0.5247	0.5207
random_forest_rs	0.8090	0.6116	0.3738	0.9326	0.4640	0.3559	0.3719	0.7352	0.4863	0.4822
gradient_boosted_trees	0.8143	0.6592	0.3323	0.9512	0.4419	0.3447	0.3739	0.7755	0.5475	0.5478
gradient_boosted_trees_rs	0.8060	0.6049	0.3542	0.9343	0.4468	0.3388	0.3566	0.7547	0.5147	0.5152
xgboost	0.8097	0.6235	0.3519	0.9397	0.4499	0.3454	0.3661	0.7613	0.5202	0.5210
xgboost_rs	0.8005	0.5754	0.3738	0.9217	0.4532	0.3378	0.3496	0.7418	0.5015	0.5019
light_gbm	0.8173	0.6576	0.3632	0.9463	0.4680	0.3686	0.3923	0.7754	0.5474	0.5478
light_gbm_rs	0.8017	0.5832	0.3617	0.9266	0.4465	0.3337	0.3478	0.7538	0.5051	0.5057

Figure 14.5: Table comparing the performance of various classifiers

For the models calibrated using the randomized search (including the `_rs` suffix in the name), we used 100 random sets of hyperparameters. As the considered problem deals with imbalanced data (the minority class is ~20%), we look at recall for performance evaluation.

It seems that the basic decision tree achieved the best recall score on the test set. This came at the cost of much lower precision than the more advanced models. That is why the F1 score (a harmonic mean of precision and recall) is the lowest for the decision tree. We can see that the default LightGBM model achieved the best F1 score on the test set.

The results by no means indicate that the more complex models are inferior—they might simply require more tuning or a different set of hyperparameters. For example, the ensemble models enforced the maximum depth of the tree (determined by the corresponding hyperparameter), while the decision tree had no such limit and it reached the depth of 37. The more advanced the model, the more effort it requires to “get it right.”

There are many different ensemble classifiers available to experiment with. Some of the possibilities include:

- AdaBoost—the first boosting algorithm.
- Extremely Randomized Trees—this algorithm offers improved randomness as compared to Random Forests. Similar to Random Forest, a random subset of features is considered when making a split. However, instead of looking for the most discriminative thresholds, the thresholds are drawn at random for each feature. Then, the best of these random thresholds is picked as the splitting rule. Such an approach usually allows us to reduce the variance of the model, while slightly increasing its bias.
- CatBoost—another boosting algorithm (developed by Yandex) that puts a high emphasis on handling categorical features and achieving high performance with little hyperparameter tuning.
- NGBoost—at a very high level, this model introduces uncertainty estimation into the gradient boosting by using the natural gradient.

- Histogram-based gradient boosting—a variant of gradient boosted trees available in `scikit-learn` and inspired by LightGBM. They accelerate the training procedure by discretizing (binning) the continuous features into a predetermined number of unique values.



While some algorithms have introduced certain features first, the other popular implementations of gradient boosted trees often receive those as well. An example might be the histogram-based approach to discretizing continuous features. While it was introduced in LightGBM, it was later added to XGBoost as well. The same goes for the leaf-wise approach to growing trees.

See also

We present additional resources on the algorithms mentioned in this recipe:

- Breiman, L. 2001. “Random Forests.” *Machine Learning* 45(1): 5–32.
- Chen, T., & Guestrin, C. 2016, August. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd international conference on knowledge discovery and data mining*, 785–794. ACM.
- Duan, T., Anand, A., Ding, D. Y., Thai, K. K., Basu, S., Ng, A., & Schuler, A. 2020, November. Ngboost: Natural gradient boosting for probabilistic prediction. In *International Conference on Machine Learning*, 2690–2700. PMLR.
- Freund, Y., & Schapire, R. E. 1996, July. Experiments with a new boosting algorithm. In *International Conference on Machine Learning*, 96: 148–156.
- Freund, Y., & Schapire, R. E. 1997. “A decision-theoretic generalization of on-line learning and an application to boosting.” *Journal of Computer and System Sciences*, 55(1), 119–139.
- Friedman, J. H. 2001. “Greedy function approximation: a gradient boosting machine.” *Annals of Statistics*, 29(5): 1189–1232.
- Friedman, J. H. 2002. “Stochastic gradient boosting.” *Computational Statistics & Data Analysis*, 38(4): 367–378.
- Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. 2017. “Lightgbm: A highly efficient gradient boosting decision tree.” In *Neural Information Processing Systems*.
- Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. 2018. CatBoost: unbiased boosting with categorical features. In *Neural information Processing Systems*.

Exploring alternative approaches to encoding categorical features

In the previous chapter, we introduced one-hot encoding as the standard solution for encoding categorical features so that they can be understood by ML algorithms. To recap, one-hot encoding converts categorical variables into several binary columns, where a value of 1 indicates that the row belongs to a certain category, and a value of 0 indicates otherwise.

The biggest drawback of that approach is the quickly expanding dimensionality of our dataset. For example, if we had a feature indicating from which of the US states the observation originates, one-hot encoding of this feature would result in the creation of 50 (or 49 if we dropped the reference value) new columns.

Some other issues with one-hot encoding include:

- Creating that many Boolean features introduces sparsity to the dataset, which decision trees don't handle well.
- Decision trees' splitting algorithm treats all the one-hot-encoded dummies as independent features. It means that when a tree makes a split using one of the dummy variables, the gain in purity per split is small. Thus, the tree is not likely to select one of the dummy variables closer to its root.
- Connected to the previous point, continuous features will have higher feature importance than one-hot encoding dummy variables, as a single dummy can only bring a fraction of its respective categorical feature's total information into the model.
- Gradient boosted trees don't handle high-cardinality features well, as the base learners have limited depth.

When dealing with a continuous variable, the splitting algorithm induces an ordering of the samples and can split that ordered list anywhere. A binary feature can only be split in one place, while a categorical feature with k unique categories can be split in $(2^k)/2 - 1$ ways.

We illustrate the advantage of the continuous features with an example. Assume that the splitting algorithm splits a continuous feature at a value of 10 into two groups: "below 10" and "10 and above." In the next split, it can further split any of the two groups, for example, "below 6" and "6 and above." That is not possible for a binary feature, as we can at most use it to split the groups once into "yes" or "no" groups. *Figure 14.6* illustrates potential differences between decision trees created with or without one-hot encoding.

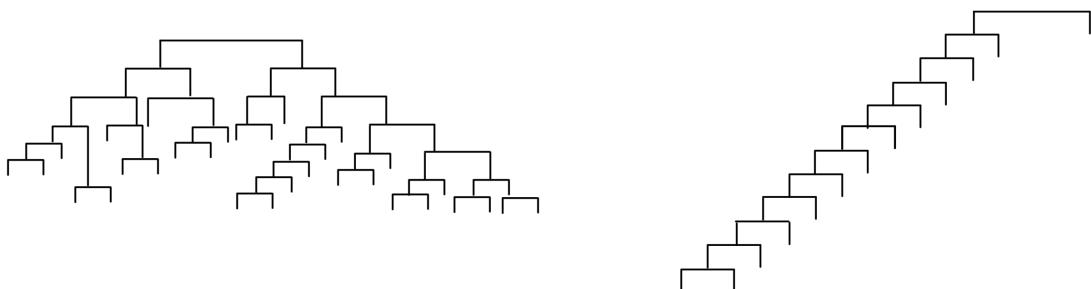


Figure 14.6: Example of a dense decision tree without one-hot encoding (on the left) and a sparse decision tree with one-hot encoding (on the right)

Those drawbacks, among others, led to the development of a few alternative approaches to encoding categorical features. In this recipe, we introduce three of them.

The first one is called **target encoding** (also known as mean encoding). In this approach, the following transformation is applied to a categorical feature, depending on the type of the target variable:

- Categorical target—a feature is replaced with a blend of the posterior probability of the target given a certain category and the prior probability of the target over all the training data.
- Continuous target—a feature is replaced with a blend of the expected value of the target given a certain category and the expected value of the target over all the training data.

In practice, the simplest scenario assumes that each category in the feature is replaced with the mean of the target value for that category. *Figure 14.7* illustrates this.

original data		posterior probabilities		encoded data		
category	target	category	target mean	category	target encoding	target
a	1	a	0.5	a	0.5	1
a	0	b	0.67	a	0.5	0
b	1			b	0.67	1
b	1			b	0.67	1
b	0			b	0.67	0

Figure 14.7: Example of target encoding

Target encoding results in a more direct representation of the relationship between the categorical feature and the target, while not adding any new columns. That is why it is a very popular technique in data science competitions.

Unfortunately, it is not a silver bullet to encoding categorical features and comes with its disadvantages:

- The approach is very prone to overfitting. That is why it assumes blending/smoothing of the category mean with the global mean. We should be especially cautious when some categories are very infrequent.
- Connected to the risk of overfitting, we are effectively leaking target information into the features.

In practice, target encoding works quite well when we have high-cardinality features and are using some form of gradient boosted trees as our machine learning model.

The second approach we cover is called **Leave One Out Encoding (LOOE)** and it is very similar to target encoding. It attempts to reduce overfitting by excluding the current row's target value when calculating the average of the category. This way, the algorithm avoids row-wise leakage. Another consequence of this approach is that the same category in multiple observations can have a different value in the encoded column. *Figure 14.8* illustrates this.

original data		encoded data		
category	target	category	leave one out encoding	target
a	1	a	0	1
a	0	a	1	0
b	1	b	0.5	1
b	1	b	0.5	1
b	0	b	1	0

Figure 14.8: Example of Leave One Out Encoding

With LOOE, the ML model is exposed not only to the same value for each encoded category (as in target encoding) but to a range of values. That is why it should learn to generalize better.

The last of the considered encodings is called **Weight of Evidence (WoE)** encoding. This one is especially interesting, as it originates from the credit scoring world, where it was employed to improve the probability of default estimates. It was used to separate customers who defaulted on the loan from those who paid it back successfully.



Weight of Evidence evolved from logistic regression. Another useful metric with the same origin as WoE is called **Information Value (IV)**. It measures how much information a feature provides for the prediction. To put it a bit differently, it helps rank variables based on their importance in the model.

The weight of evidence indicates the predictive power of an independent variable in relation to the target. In other words, it measures how much the evidence supports or undermines a hypothesis. It is defined as the natural logarithm of the odds ratio:

$$WoE = \ln \left(\frac{\% \text{ of good customers in a group}}{\% \text{ of bad customers in a group}} \right)$$

Figure 14.9 illustrates the calculations.

original data		Weight of Evidence						
category	target	category	# total	# good	# bad	% good	% bad	WoE
a	1	a	2	1	1	0,5000	0,3333	0,4055
a	0	b	3	1	2	0,5000	0,6667	-0,2877
b	1	total		5	2	3		
b	1							
b	0							

Figure 14.9: Example of the WoE encoding

The fact that the encoding originates from credit scoring does not mean that it is only usable in such cases. We can generalize the good customers as the non-event or negative class, and the bad customers as the event or positive class. One of the restrictions of the approach is that, in contrast to the previous two, it can only be used with a binary categorical target.



WoE was also historically used to encode categorical features as well. For example, in a credit scoring dataset, we could bin a continuous feature like age into discrete bins: 20–29, 30–39, 40–49, and so on, and only then calculate the WoE for those categories. The number of bins chosen for the encoding depends on the use case and the feature's distribution.

In this recipe, we show how to use those three encoders in practice using the default dataset we have already used before.

Getting ready

In this recipe, we use the pipeline we have used in the previous recipes. As the estimator, we use the Random Forest classifier. For your convenience, we reiterate all the required steps in the Jupyter notebook accompanying this chapter.

The Random Forest pipeline with one-hot encoded categorical features resulted in the test set's recall of 0.3542. We will try to improve upon this score with alternative approaches to encoding categorical features.

How to do it...

Execute the following steps to fit the ML pipelines with various categorical encoders:

- Import the libraries:

```
import category_encoders as ce
from sklearn.base import clone
```

2. Fit the pipeline using target encoding:

```

pipeline_target_enc = clone(rf_pipeline)
pipeline_target_enc.set_params(
    preprocessor_categorical_cat_encoding=ce.TargetEncoder()
)

pipeline_target_enc.fit(X_train, y_train)
target_enc_perf = performance_evaluation_report(
    pipeline_target_enc, X_test,
    y_test, labels=LABELS,
    show_plot=True,
    show_pr_curve=True
)
print(f'Recall: {target_enc_perf["recall"]:.4f}')

```

Executing the snippet generates the following plot:

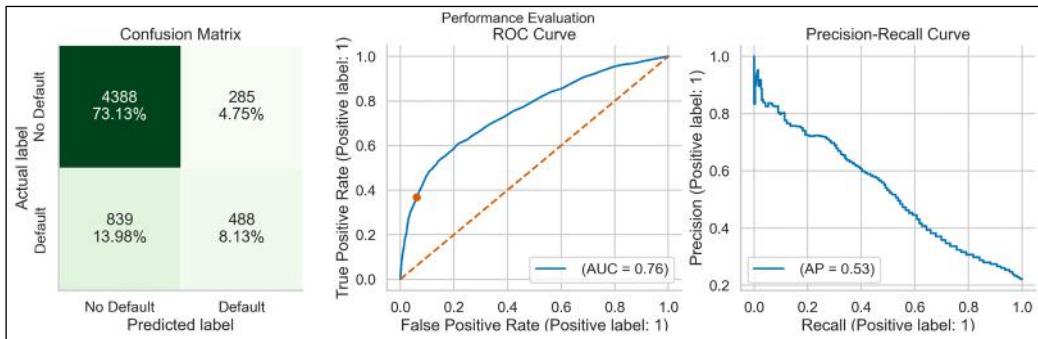


Figure 14.10: Performance evaluation of the pipeline with target encoding

The recall obtained using this pipeline is equal to 0.3677. This improves the score by slightly over 1 p.p.

3. Fit the pipeline using Leave One Out Encoding:

```

pipeline_loo_enc = clone(rf_pipeline)
pipeline_loo_enc.set_params(
    preprocessor_categorical_cat_encoding=ce.LeaveOneOutEncoder()
)

pipeline_loo_enc.fit(X_train, y_train)

```

```

loo_enc_perf = performance_evaluation_report(
    pipeline_loo_enc, X_test,
    y_test, labels=LABELS,
    show_plot=True,
    show_pr_curve=True
)
print(f'Recall: {loo_enc_perf["recall"]:.4f}')

```

Executing the snippet generates the following plot:

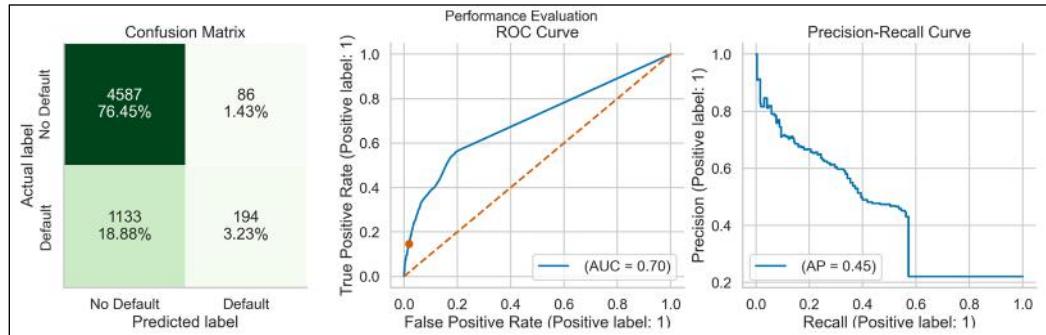


Figure 14.11: Performance evaluation of the pipeline with Leave One Out Encoding

The recall obtained using this pipeline is equal to 0.1462 , which is significantly worse than the target encoding approach.

- Fit the pipeline using Weight of Evidence encoding:

```

pipeline_woe_enc = clone(rf_pipeline)
pipeline_woe_enc.set_params(
    preprocessor_categorical_cat_encoding=ce.WOEEncoder()
)

pipeline_woe_enc.fit(X_train, y_train)
woe_enc_perf = performance_evaluation_report(
    pipeline_woe_enc, X_test,
    y_test, labels=LABELS,
    show_plot=True,
    show_pr_curve=True
)
print(f'Recall: {woe_enc_perf["recall"]:.4f}')

```

Executing the snippet generates the following plot:

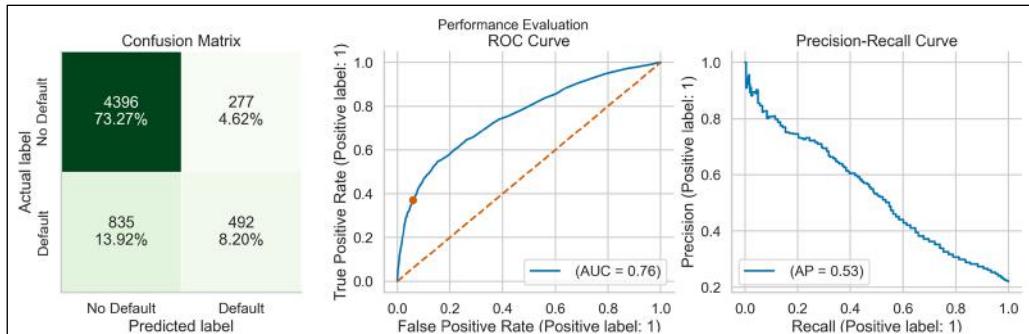


Figure 14.12: Performance evaluation of the pipeline with Weight of Evidence encoding

The recall obtained using this pipeline is equal to 0.3708 , which is a small improvement over target encoding.

How it works...

First, we executed the code from the *Getting ready* section, that is, instantiated the pipeline with one-hot encoding and Random Forest as the classifier.

After importing the libraries, we cloned the entire pipeline using the `clone` function. Then, we used the `set_params` method to replace the `OneHotEncoder` with `TargetEncoder`. Just as when tuning the hyperparameters of a pipeline, we had to use the same double underscore notation to access the particular element of the pipeline. The encoder was located under `preprocessor_categorical_cat_encoding`. Then, we fitted the pipeline using the `fit` method and printed the evaluation scores using the `performance_evaluation_report` helper function.

As we have mentioned in the introduction, target encoding is prone to overfitting. That is why instead of simply replacing the categories with the corresponding averages, the algorithm is capable of blending the posterior probabilities with the prior probability (global average). We can control the blending with two hyperparameters: `min_samples_leaf` and `smoothing`.

In *Steps 3 and 4*, we followed the very same steps as with target encoding, but we replaced the encoder with `LeaveOneOutEncoder` and `WOEEncoder` respectively.

Just as with target encoding, the other encoders use the target to build the encoding and are thus prone to overfitting. Fortunately, they also offer certain measures to prevent that from happening.

In the case of LOOE, we can add normally distributed noise to the encodings in order to reduce overfitting. We can control the standard deviation of the Normal distribution used for generating the noise with the `sigma` argument. It is worth mentioning that the random noise is added to the training data only, and the transformation of the test set is not impacted. Just by adding the random noise to our pipeline (`sigma = 0.05`), we can improve the measured recall score from 0.1462 to around 0.35 (depending on random number generation).

Similarly, we can add random noise for the WoE encoder. We control the noise with the `randomized` (Boolean flag) and `sigma` (standard deviation of the Normal distribution) arguments. Additionally, there is the `regularization` argument, which prevents errors caused by division by zero.

There's more...

Encoding categorical variables is a very broad area of active research, and every now and then new approaches to it are being published. Before changing the topic, we would also like to discuss a couple of related concepts.

Handling data leakage with k-fold target encoding

We have already mentioned a few approaches to reducing the overfitting problem of the target encoder. A very popular solution among Kaggle practitioners is to use *k*-fold target encoding. The idea is similar to *k*-fold cross-validation and it allows us to use all the training data we have. We start by dividing the data into *k* folds—they can be stratified or purely random, depending on the use case. Then, we replace the observations present in the *l*-th fold with the target's mean calculated using all the folds except the *l*-th one. This way, we are not leaking the target from the observations within the same fold.



An inquisitive reader might have noticed that the LOOE is a special case of *k*-fold target encoding, in which *k* is equal to the number of observations in the training dataset.

Even more encoders

The `category_encoders` library offers almost 20 different encoding transformers for categorical features. Aside from the ones we have already mentioned, you might want to explore the following:

- **Ordinal encoding**—very similar to label encoding; however, it ensures that the encoding retains the ordinal nature of the feature. For example, the hierarchy of bad < neutral < good is preserved.
- **Count encoder** (frequency encoder)—each category of a feature is mapped to the number of observations belonging to that category.
- **Sum encoder**—compares the mean of the target for a given category to the overall average of the target.
- **Helmert encoder**—compares the mean of a certain category to the mean of the subsequent levels. If we had categories [A, B, C], the algorithm would first compare A to B and C and then B to C alone. This kind of encoding is useful in situations in which the levels of the categorical feature are ordered, for example, from lowest to highest.
- **Backward difference encoder**—similar to the Helmert encoder, with the difference that it compares the mean of the current category to the mean of the previous one.
- **M-estimate encoder**—a simplified version of the target encoder, which has only one tunable parameter (responsible for the strength of regularization).

- **James-Stein encoder**—a variant of target encoding that aims to improve the estimation of the category's mean by shrinking it toward the central/global mean. Its single hyperparameter is responsible for the strength of shrinkage (this means the same as regularization in this context)—the bigger the value of the hyperparameter, the bigger the weight of the global mean (which might lead to underfitting). On the other hand, reducing the hyperparameter's value might lead to overfitting. The best value is usually determined by cross-validation. The approach's biggest disadvantage is that the James-Stein estimator is defined only for Normal distribution, which is not the case for any binary classification problem.
- **Binary encoder**—converts a category into binary digits and each one is provided a separate column. Thanks to this encoding, we generate far fewer columns than with OHE. To illustrate, for a categorical feature with 100 unique categories, binary encoding just needs to create 7 features, instead of 100 in the case of OHE.
- **Hashing encoder**—uses a hashing function (often used in data encryption) to transform the categorical features. The outcome is similar to OHE, but with fewer features (we can control that with the encoder's hyperparameters). It has two significant disadvantages. First, the encoding results in information loss, as the algorithm transforms the full set of available categories into fewer features. The second issue is called collision and it occurs as we are transforming a potentially high number of categories into a smaller set of features. Then, different categories could be represented by the same hash values.
- **Catboost encoder**—an improved variant of Leave One Out Encoding, which aims to overcome the issues of target leakage.

See also

- Micci-Barreca, D. 2001. “A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems.” *ACM SIGKDD Explorations Newsletter* 3(1): 27–32.

Investigating different approaches to handling imbalanced data

A very common issue when working with classification tasks is that of **class imbalance**, that is, when one class is highly outnumbered in comparison to the second one (this can also be extended to multi-class cases). In general, we are dealing with imbalance when the ratio of the two classes is not 1:1. In some cases, a delicate imbalance is not that big of a problem, but there are industries/problems in which we can encounter ratios of 100:1, 1000:1, or even more extreme.

Dealing with highly imbalanced classes can result in the poor performance of ML models. That is because most of the algorithms implicitly assume balanced distribution of classes. They do so by aiming to minimize the overall prediction error, to which the minority class by definition contributes very little. As a result, classifiers trained on imbalanced data are biased toward the majority class.

One of the potential solutions to dealing with class imbalance is to resample the data. On a high level, we can either undersample the majority class, oversample the minority class, or combine the two approaches. However, that is just the general idea. There are many ways to approach resampling and we describe a few selected methods below.



When working with resampling techniques, we only resample the training data! The test data stays intact.

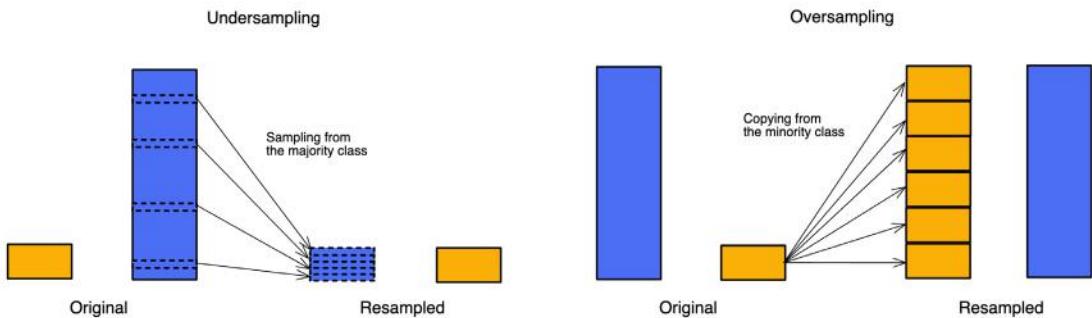


Figure 14.13: Undersampling of the majority class and oversampling of the minority class

The simplest approach to undersampling is called **random undersampling**. In this approach, we undersample the majority class, that is, draw random samples (by default, without replacement) from the majority class until the classes are balanced (with a ratio of 1:1 or any other desired ratio). The biggest issue of this method is the information loss caused by discarding vast amounts of data, often the majority of the entire training dataset. As a result, a model trained on undersampled data can achieve lower performance. Another possible implication is a biased classifier with an increased number of false positives, as the distribution of the training and test sets is not the same after resampling.

Analogically, the simplest approach to oversampling is called **random oversampling**. In this approach, we sample multiple times with replacement from the minority class, until the desired ratio is achieved. This method often outperforms random undersampling, as there is no information loss caused by discarding training data. However, random oversampling comes with the danger of overfitting, caused by replicating observations from the minority class.

Synthetic Minority Oversampling Technique (SMOTE) is a more advanced oversampling algorithm that creates new, synthetic observations from the minority class. This way, it overcomes the previously mentioned problem of overfitting.

To create the synthetic samples, the algorithm picks an observation from the minority class, identifies its k -nearest neighbors (using the k -NN algorithm), and then creates new observations on the lines connecting (interpolating) the observation to the nearest neighbors. Then, the process is repeated for other minority observations until the classes are balanced.

Aside from reducing the problem of overfitting, SMOTE causes no loss of information, as it does not discard observations belonging to the majority class. However, SMOTE can accidentally introduce more noise to the data and cause overlapping of classes. This is because while creating the synthetic observations, it does not take into account the observations from the majority class. Additionally, the algorithm is not very effective for high-dimensional data (due to the curse of dimensionality). Lastly, the basic variant of SMOTE is only suitable for numerical features. However, SMOTE's extensions (mentioned in the *There's more...* section) can handle categorical features as well.

The last of the considered oversampling techniques is called **Adaptive Synthetic Sampling (ADASYN)** and it is a modification of the SMOTE algorithm. In ADASYN, the number of observations to be created for a certain minority point is determined by a density distribution (instead of a uniform weight for all points, as in SMOTE). This is how ADASYN's adaptive nature enables it to generate more synthetic samples for observations that come from hard-to-learn neighborhoods. For example, a minority observation is hard to learn if there are many majority class observations with very similar feature values. It is easier to imagine that scenario in the case of only two features. Then, in a scatterplot, such a minority class observation might simply be surrounded by many of the majority class observations.

There are two additional elements worth mentioning:

- In contrast to SMOTE, the synthetic points are not limited to linear interpolation between two points. They can also lie on a plane created by three or more observations.
- After creating the synthetic observations, the algorithm adds a small random noise to increase the variance, thus making the samples more realistic.

Potential drawbacks of ADASYN include:

- A possible decrease in precision (more false positives) of the algorithm caused by its adaptability. This means that the algorithm might generate more observations in the areas with high numbers of observations from the majority class. Such synthetic data might be very similar to those majority class observations, potentially resulting in more false positives.
- Struggling with sparsely distributed minority observations. Then, a neighborhood can contain only one or very few points.

Resampling is not the only potential solution to the problem of imbalanced classes. Another one is based on adjusting the class weights, thus putting more weight on the minority class. In the background, the class weights are incorporated into calculating the loss function. In practice, this means that misclassifying observations from the minority class increases the value of the loss function significantly more than in the case of misclassifying the observations from the majority class.

In this recipe, we show an example of a credit card fraud problem, where the fraudulent class is observed in only 0.17% of the entire sample. In such cases, gathering more data (especially of the fraudulent class) might simply not be feasible, and we need to resort to other techniques that can help us in improving the models' performance.

Getting ready

Before proceeding to the coding part, we provide a brief description of the dataset selected for this exercise. You can download the dataset from Kaggle (link in the *See also* section).

The dataset contains information about credit card transactions made over a period of two days in September 2013 by European cardholders. Due to confidentiality, almost all features (28 out of 30) were anonymized by using **Principal Components Analysis (PCA)**. The only two features with clear interpretation are **Time** (seconds elapsed between each transaction and the first one in the dataset) and **Amount** (the transaction's amount).

Lastly, the dataset is highly imbalanced and the positive class is observed in 0.173% of all transactions. To be precise, out of 284,807 transactions, 492 were identified as fraudulent.

How to do it...

Execute the following steps to investigate different approaches to handling class imbalance:

1. Import the libraries:

```
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import RobustScaler

from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
from imblearn.ensemble import BalancedRandomForestClassifier

from chapter_14_utils import performance_evaluation_report
```

2. Load and prepare data:

```
RANDOM_STATE = 42

df = pd.read_csv("../Datasets/credit_card_fraud.csv")
X = df.copy().drop(columns=["Time"])
y = X.pop("Class")

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=RANDOM_STATE
)
```

Using `y.value_counts(normalize=True)` we can confirm that the positive class is observed in 0.173% of the observations.

3. Scale the features using RobustScaler:

```
robust_scaler = RobustScaler()  
X_train = robust_scaler.fit_transform(X_train)  
X_test = robust_scaler.transform(X_test)
```

4. Train the baseline model:

```
rf = RandomForestClassifier(  
    random_state=RANDOM_STATE, n_jobs=-1  
)  
rf.fit(X_train, y_train)
```

5. Undersample the training data and train a Random Forest classifier:

```
rus = RandomUnderSampler(random_state=RANDOM_STATE)  
X_rus, y_rus = rus.fit_resample(X_train, y_train)  
  
rf.fit(X_rus, y_rus)  
rf_rus_perf = performance_evaluation_report(rf, X_test, y_test)
```

After random undersampling, the ratio of the classes is as follows: {0: 394, 1: 394}.

6. Oversample the training data and train a Random Forest classifier:

```
ros = RandomOverSampler(random_state=RANDOM_STATE)  
X_ros, y_ros = ros.fit_resample(X_train, y_train)  
  
rf.fit(X_ros, y_ros)  
rf_ros_perf = performance_evaluation_report(rf, X_test, y_test)
```

After random oversampling, the ratio of the classes is as follows: {0: 227451, 1: 227451}.

7. Oversample the training data using SMOTE:

```
smote = SMOTE(random_state=RANDOM_STATE)  
X_smote, y_smote = smote.fit_resample(X_train, y_train)  
  
rf.fit(X_smote, y_smote)  
rf_smote_perf = performance_evaluation_report(  
    rf, X_test, y_test,  
)
```

After oversampling with SMOTE, the ratio of the classes is as follows: {0: 227451, 1: 227451}.

8. Oversample the training data using ADASYN:

```
adasyn = ADASYN(random_state=RANDOM_STATE)
X_adasyn, y_adasyn = adasyn.fit_resample(X_train, y_train)

rf.fit(X_adasyn, y_adasyn)
rf_adasyn_perf = performance_evaluation_report(
    rf, X_test, y_test,
)
```

After oversampling with ADASYN, the ratio of the classes is as follows:
{0: 227451, 1: 227449}.

9. Use sample weights in the Random Forest classifier:

```
rf_cw = RandomForestClassifier(random_state=RANDOM_STATE,
                               class_weight="balanced",
                               n_jobs=-1)
rf_cw.fit(X_train, y_train)

rf_cw_perf = performance_evaluation_report(
    rf_cw, X_test, y_test,
)
```

10. Train the `BalancedRandomForestClassifier`:

```
balanced_rf = BalancedRandomForestClassifier(
    random_state=RANDOM_STATE
)

balanced_rf.fit(X_train, y_train)
balanced_rf_perf = performance_evaluation_report(
    balanced_rf, X_test, y_test,
)
```

11. Train the `BalancedRandomForestClassifier` with balanced classes:

```
balanced_rf_cw = BalancedRandomForestClassifier(
    random_state=RANDOM_STATE,
    class_weight="balanced",
    n_jobs=-1
)

balanced_rf_cw.fit(X_train, y_train)
```

```

balanced_rf_cw_perf = performance_evaluation_report(
    balanced_rf_cw, X_test, y_test,
)

```

12. Combine the results in a DataFrame:

```

performance_results = {
    "random_forest": rf_perf,
    "undersampled_rf": rf_rus_perf,
    "oversampled_rf": rf_ros_perf,
    "smote": rf_smote_perf,
    "adasyn": rf_adasyn_perf,
    "random_forest_cw": rf_cw_perf,
    "balanced_random_forest": balanced_rf_perf,
    "balanced_random_forest_cw": balanced_rf_cw_perf,
}
pd.DataFrame(performance_results).round(4).T

```

Executing the snippet prints the following table:

	accuracy	precision	recall	specificity	f1_score	cohens_kappa	matthews_corr_coeff	roc_auc	pr_auc	average_precision
random_forest	0.9996	0.9419	0.8265	0.9999	0.8804	0.8802	0.8821	0.9528	0.8761	0.8715
undersampled_rf	0.9672	0.0457	0.9082	0.9673	0.0870	0.0840	0.1996	0.9780	0.7493	0.6894
oversampled_rf	0.9996	0.9506	0.7857	0.9999	0.8603	0.8601	0.8640	0.9526	0.8694	0.8650
smote	0.9995	0.8817	0.8367	0.9998	0.8586	0.8584	0.8587	0.9630	0.8791	0.8779
adasyn	0.9994	0.8511	0.8163	0.9998	0.8333	0.8331	0.8332	0.9731	0.8661	0.8635
random_forest_cw	0.9995	0.9610	0.7551	0.9999	0.8457	0.8455	0.8517	0.9580	0.8572	0.8483
balanced_random_forest	0.9738	0.0567	0.9082	0.9740	0.1067	0.1038	0.2233	0.9761	0.7800	0.7253
balanced_random_forest_cw	0.9864	0.1025	0.8878	0.9866	0.1837	0.1812	0.2990	0.9780	0.7240	0.6759

Figure 14.14: Performance evaluation metrics of the various approaches to dealing with imbalanced data

In Figure 14.14 we can see the performance evaluation of various approaches we have tried in this recipe. As we are dealing with a highly imbalanced problem (the positive class accounts for 0.17% of all the observations), we can clearly observe the case of the **accuracy paradox**. Many models have an accuracy of $\approx 99.9\%$, but they still fail to detect fraudulent cases, which are the most important ones.



The accuracy paradox refers to a case in which inspecting accuracy as the evaluation metric creates the impression of having a very good classifier (a score of 90%, or even 99.9%), while in reality it simply reflects the distribution of the classes.

Taking that into consideration, we compare the performance of the models using metrics that account for that. While looking at precision, the best performing approach is Random Forest with class weights. When considering recall as the most important metric, the best performing approach is either undersampling followed by a Random Forest model or a Balanced Random Forest model. In terms of the F1 score, the best approach seems to be the vanilla Random Forest model.

It is also important to mention that no hyperparameter tuning was performed, which could potentially improve the performance of all of the approaches.

How it works...

After importing the libraries, we loaded the credit card fraud dataset from a CSV file. In the same step, we additionally dropped the `Time` feature, separated the target from the features using the `pop` method, and created an 80–20 stratified train-test split. It is crucial to remember to use stratification when dealing with imbalanced classes.

In this recipe, we only focused on working with imbalanced data. That is why we did not cover any EDA, feature engineering, and so on. As all the features were numerical, we did not have to carry out any special encoding.

The only preprocessing step we did was to scale all the features using `RobustScaler`. While Random Forest does not require explicit feature scaling, some of the rebalancing approaches use k -NN under the hood. And for such distance-based algorithms, the scale does matter. We fitted the scaler using only the training data and then transformed both the training and test sets.

In *Step 4*, we fitted a vanilla Random Forest model, which we used as a benchmark for the more complex approaches.

In *Step 5*, we used the `RandomUnderSampler` class from the `imblearn` library to randomly undersample the majority class in order to match the size of the minority sample. Conveniently, classes from `imblearn` follow `scikit-learn`'s API style. That is why we had to first define the class with the arguments (we only set the `random_state`). Then, we applied the `fit_resample` method to obtain the undersampled data. We reused the Random Forest object to train the model on the undersampled data and stored the results for later comparison.

Step 6 is analogical to *Step 5*, with the only difference being the use of the `RandomOverSampler` to randomly oversample the minority class in order to match the size of the majority class.

In *Step 7* and *Step 8*, we applied the SMOTE and ADASYN variants of oversampling. As the `imblearn` library makes it very easy to apply different sampling methods, we will not go deeper into the description of the process.



In all the mentioned resampling methods, we can actually specify the desired ratio between classes by passing a float to the `sampling_strategy` argument. The number represents the desired ratio of the number of observations in the minority class over the number of observations in the majority class.

In *Step 9*, instead of resampling the training data, we used the `class_weight` hyperparameter of the `RandomForestClassifier` to account for the class imbalance. By passing “`balanced`”, the algorithm automatically assigns weights inversely proportional to class frequencies in the training data.



There are different possible approaches to using the `class_weight` hyperparameter. Passing "balanced_subsample" results in a similar weights assignment as in "balanced"; however, the weights are computed based on the bootstrap sample for every tree. Alternatively, we can pass a dictionary containing the desired weights. One way of determining the weights can be by using the `compute_class_weight` function from `sklearn.utils.class_weight`.

The `imblearn` library also features some modified versions of popular classifiers. In *Steps 10 and 11*, we used a modified Random Forest classifier, that is, **Balanced Random Forest**. The difference is that in Balanced Random Forest the algorithm randomly undersamples each bootstrapped sample to balance the classes. In practical terms, its API is virtually the same as in the vanilla `scikit-learn` implementation (including the tunable hyperparameters).

In the last step, we combined all the results into a single DataFrame and displayed the results.

There's more...

In this recipe, we presented only some of the available resampling methods. Below, we list a few more possibilities.

Undersampling:

- **NearMiss**—the name refers to a collection of undersampling approaches that are essentially heuristic rules based on the Nearest Neighbors algorithm. They base the selection of the observations from the majority class to keep on the distance between the observations from the majority and minority classes. The rest is removed in order to balance the classes. For example, the NearMiss-1 method selects observations from the majority class that have the smallest average distance to the three closest observations from the minority class.
- **Edited Nearest Neighbors**—this approach removes any majority class observation whose class is different from the class of at least two of its three nearest neighbors. The underlying idea is to remove the instances from the majority class that are near the boundary of classes.
- **Tomek links**—in this undersampling heuristic we first identify all the pairs of observations that are nearest to each other (they are the nearest neighbors) but belong to different classes. Such pairs are called Tomek links. Then, from those pairs, we remove the observations that belong to the majority class. The underlying idea is that by removing those observations from the Tomek link we increase the class separation.

Oversampling:

- **SMOTE-NC (Synthetic Minority Oversampling Technique for Nominal and Continuous)**—a variant of SMOTE suitable for a dataset containing both numerical and categorical features. The vanilla SMOTE can create illogical values for one-hot-encoded features.
- **Borderline SMOTE**—this variant of the SMOTE algorithm will create new, synthetic observations along the decision boundary between the two classes, as those are more prone to being misclassified.

- **SVM SMOTE**—a variant of SMOTE in which an SVM algorithm is used to indicate which observations to use for generating new synthetic observations.
- **K-means SMOTE**—in this approach, we first apply k -means clustering to identify clusters with a high proportion of minority class observations. Then, the vanilla SMOTE is applied to the selected clusters and each of those clusters will have new synthetic observations.

Alternatively, we could combine the undersampling and oversampling approaches. The underlying idea is to first use an oversampling method to create duplicate or artificial observations and then use an undersampling method to reduce the noise or remove unnecessary observations.

For example, we could first oversample the data with SMOTE and then undersample it using random undersampling. `imbalanced-learn` offers two combined resamplers—SMOTE followed by Tomek links or Edited Nearest Neighbours.

In this recipe, we have only covered a small selection of the available approaches. Before changing topics, we wanted to mention some general notes on tackling problems with imbalanced classes:

- Do not apply under/oversampling on the test set.
- For evaluating problems with imbalanced data, use metrics that account for class imbalance, such as precision, recall, F1 score, Cohen's kappa, or the PR-AUC.
- Use stratification when creating folds for cross-validation.
- Introduce under-/oversampling during cross-validation, not before. Doing so before leads to overestimating the model's performance!
- When creating pipelines with resampling using the `imbalanced-learn` library, we also need to use the `imbalanced-learn` variants of the pipeline. This is because the resamplers use the `fit_resample` method instead of the `fit_transform` required by `scikit-learn`'s pipelines.
- Consider framing the problem differently. For example, instead of a classification task, we could treat it as an anomaly detection problem. Then, we could use different techniques, for example, **isolation forest**.
- Experiment with selecting a different probability threshold than the default 50% to potentially tune the performance. Instead of rebalancing the dataset, we can use the model trained using the imbalanced dataset to plot the false positive and false negative rates as a function of the decision threshold. Then, we can choose the threshold that results in the performance that best suits our needs.



We use the decision threshold to determine over which probability or score (a classifier's output) we consider that the given observation belongs to the positive class. By default, that is 0.5.

See also

The dataset we have used in this recipe is available on Kaggle:

- <https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud>

Additional resources are available here:

- Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. 2002. “SMOTE: synthetic minority oversampling technique.” *Journal of artificial intelligence research* 16: 321–357.
- Chawla, N. V. 2009. “Data mining for imbalanced datasets: An overview.” *Data mining and knowledge discovery handbook*: 875–886.
- Chen, C., Liaw, A., & Breiman, L. 2004. “Using random forest to learn imbalanced data.” *University of California, Berkeley* 110: 1–12.
- Elor, Y., & Averbuch-Elor, H. 2022. “To SMOTE, or not to SMOTE?” *arXiv preprint arXiv:2201.08528*.
- Han, H., Wang, W. Y., & Mao, B. H. 2005, August. Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning. In *International conference on intelligent computing*, 878–887. Springer, Berlin, Heidelberg.
- He, H., Bai, Y., Garcia, E. A., & Li, S. 2008, June. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*, 1322–1328. IEEE.
- Le Borgne, Y.-A., Siblini, W., Lebichot, B., & Bontempi, G. 2022. Reproducible Machine Learning for Credit Card Fraud Detection – Practical Handbook.
- Liu, F. T., Ting, K. M., & Zhou, Z. H. 2008, December. Isolation forest. In *2008 Eighth Ieee International Conference On Data Mining*, 413–422. IEEE.
- Mani, I., & Zhang, I. 2003, August. kNN approach to unbalanced data distributions: a case study involving information extraction. In *Proceedings of workshop on learning from imbalanced datasets*, 126: 1–7. ICML.
- Nguyen, H. M., Cooper, E. W., & Kamei, K. 2009, November. Borderline over-sampling for imbalanced data classification. In *Proceedings: Fifth International Workshop on Computational Intelligence & Applications*, 2009(1): 24–29. IEEE SMC Hiroshima Chapter.
- Pozzolo, A.D. et al. 2015. Calibrating Probability with Undersampling for Unbalanced Classification, *2015 IEEE Symposium Series on Computational Intelligence*.
- Tomek, I. (1976). Two modifications of CNN, *IEEE Transactions on Systems Man and Communications*, 6: 769-772.
- Wilson, D. L. (1972). “Asymptotic properties of nearest neighbor rules using edited data.” *IEEE Transactions on Systems, Man, and Cybernetics* 3: 408–421.

Leveraging the wisdom of the crowds with stacked ensembles

Stacking (stacked generalization) refers to a technique of creating ensembles of potentially heterogeneous machine learning models. The architecture of a stacking ensemble comprises at least two base models (known as level 0 models) and a meta-model (the level 1 model) that combines the predictions of the base models. The following figure illustrates an example with two base models.

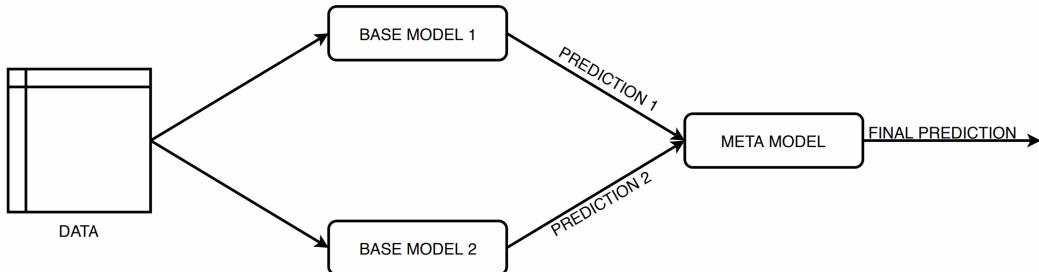


Figure 14.15: High-level schema of a stacking ensemble with two base learners

The goal of stacking is to combine the capabilities of a range of well-performing models and obtain predictions that result in a potentially better performance than any single model in the ensemble. That is possible as the stacked ensemble tries to leverage the different strengths of the base models. Because of that, the base models should often be complex and diverse. For example, we could use linear models, decision trees, various kinds of ensembles, k-nearest neighbors, support vector machines, neural networks, and so on.

Stacking can be a bit more difficult to understand than the previously covered ensemble methods (bagging, boosting, and so on) as there are at least a few variants of stacking when it comes to splitting data, handling potential overfitting, and data leakage. In this recipe, we follow the approach used in the `scikit-learn` library.

The procedure used for creating a stacked ensemble can be described in three steps. We assume that we already have representative training and test datasets.

Step 1: Train level 0 models

The essence of this step is that each of the level 0 models is trained on the full training dataset and then those models are used to generate predictions.

Then, we have a few things to consider for our ensemble. First, we have to pick what kind of predictions we want to use. For a regression problem, this is straightforward as we do not have any choice. However, when working with a classification problem we can use the predicted class or the predicted probability/score.

Second, we can either use only the predictions (whichever variant we picked before) as the features for the level 1 model or combine the original feature set with the predictions from the level 0 models. In practice, combining the features tends to work a bit better. Naturally, this heavily depends on the use case and the considered dataset.

Step 2: Train the level 1 model

The level 1 model (or the meta-model) is often quite simple and ideally can provide a smooth interpretation of the predictions made by the level 0 models. That is why linear models are often selected for this task.



The term **blending** often refers to using a simple linear model as the level 1 model. This is because the predictions of the level 1 model are then a weighted average (or blending) of the predictions made by the level 0 models.

In this *step*, the level 1 model is trained using the features from the previous step (either only the predictions or combined with the initial set of features) and some cross-validation scheme. The latter is used to select the meta-model's hyperparameters and/or the set of base models to consider for the ensemble.

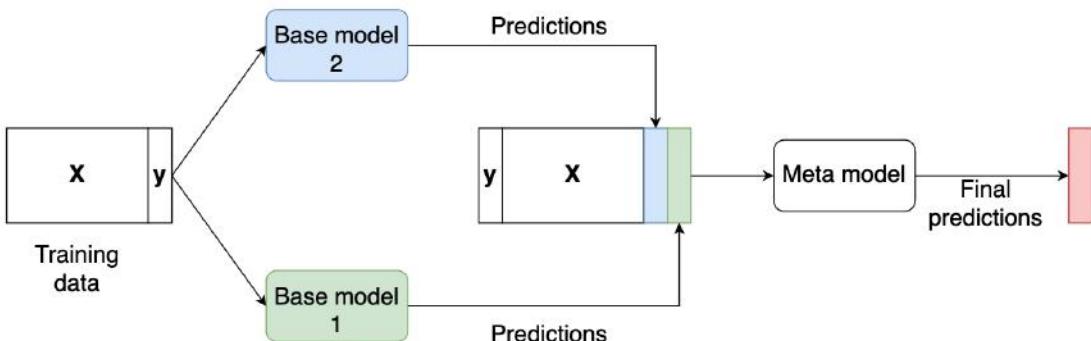


Figure 14.16: Low-level schema of a stacking ensemble with two base learners

In `scikit-learn`'s approach to stacking, we assume that any of the base models could have a tendency to overfit, either due to the algorithm itself or due to some combination of its hyperparameters. But if that is the case, it should be offset by the other base models not suffering from the same problem. That is why cross-validation is applied to tune the meta-model and not the base models as well.

After the best hyperparameters/base learners are selected, the final estimator is trained on the full training dataset.

Step 3: Make predictions on unseen data

This step is the easiest one, as we are essentially fitting all the base models to the new observations to obtain the predictions, which are then used by the meta-model to create the stacked ensemble's final predictions.

In this recipe, we create a stacked ensemble of models applied to the credit card fraud dataset.

How to do it...

Execute the following steps to create a stacked ensemble:

1. Import the libraries:

```
import pandas as pd
from sklearn.model_selection import (train_test_split,
                                      StratifiedKFold)
from sklearn.metrics import recall_score
from sklearn.preprocessing import RobustScaler

from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import StackingClassifier
```

2. Load and preprocess data:

```
RANDOM_STATE = 42

df = pd.read_csv("../Datasets/credit_card_fraud.csv")
X = df.copy().drop(columns=["Time"])
y = X.pop("Class")

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=RANDOM_STATE
)

robust_scaler = RobustScaler()
X_train = robust_scaler.fit_transform(X_train)
X_test = robust_scaler.transform(X_test)
```

3. Define a list of base models:

```
base_models = [
    ("dec_tree", DecisionTreeClassifier()),
    ("log_reg", LogisticRegression()),
    ("svc", SVC()),
    ("naive_bayes", GaussianNB())
]
```



In the accompanying Jupyter notebook, we specified the random state of all the models to which it is applicable. Here, we omitted that part for brevity.

4. Train the selected models and calculate the recall using the test set:

```
for model_tuple in base_models:  
    clf = model_tuple[1]  
    if "n_jobs" in clf.get_params().keys():  
        clf.set_params(n_jobs=-1)  
    clf.fit(X_train, y_train)  
    recall = recall_score(y_test, clf.predict(X_test))  
    print(f"{model_tuple[0]}'s recall score: {recall:.4f}")
```

Executing the snippet generates the following output:

```
dec_tree's recall score: 0.7551  
log_reg's recall score: 0.6531  
svc's recall score: 0.7041  
naive_bayes's recall score: 0.8469
```

Out of the considered models, the Naive Bayes classifier achieved the best recall on the test set.

5. Define, fit, and evaluate the stacked ensemble:

```
cv_scheme = StratifiedKFold(n_splits=5,  
                            shuffle=True,  
                            random_state=RANDOM_STATE)  
meta_model = LogisticRegression(random_state=RANDOM_STATE)  
  
stack_clf = StackingClassifier(  
    base_models,  
    final_estimator=meta_model,  
    cv=cv_scheme,  
    n_jobs=-1  
)  
stack_clf.fit(X_train, y_train)  
  
recall = recall_score(y_test, stack_clf.predict(X_test))  
print(f"The stacked ensemble's recall score: {recall:.4f}")
```

Executing the snippet generates the following output:

```
The stacked ensemble's recall score: 0.7449
```

Our stacked ensemble resulted in a worse score than the best of the individual models. However, we can try to further improve the ensemble. For example, we can allow the ensemble to use the initial features for the meta-model and replace the logistic regression meta-model with a Random Forest classifier.

6. Improve the stacking ensemble with additional features and a more complex meta-model:

```
meta_model = RandomForestClassifier(random_state=RANDOM_STATE)
stack_clf = StackingClassifier(
    base_models,
    final_estimator=meta_model,
    cv=cv_scheme,
    passthrough=True,
    n_jobs=-1
)
stack_clf.fit(X_train, y_train)
```

The second stacked ensemble achieved a recall score of `0.8571`, which is better than the best of the individual models.

How it works...

In *Step 1*, we imported the required libraries. Then, we loaded the credit card fraud dataset, separated the target from the features, dropped the `Time` feature, split the data into training and test sets (using a stratified split), and finally, scaled the data with `RobustScaler`. The transformation is not necessary for tree-based models, however; we use various classifiers (each with its own set of assumptions about the input data) as base models. For simplicity, we did not investigate different properties of the features, such as normality. Please refer to the previous recipe for more details on those processing steps.

In *Step 3*, we defined a list of base learners for the stacked ensemble. We decided to use a few simple classifiers, such as a decision tree, a Naive Bayes classifier, a support vector classifier, and logistic regression. For brevity, we will not describe the properties of the selected classifiers here.



When preparing a list of base learners, we can also provide the entire pipelines instead of just the estimators. This can come in handy when only some of the ML models require dedicated preprocessing of the features, such as scaling or encoding categorical variables.

In *Step 4*, we iterated over the list of classifiers, fitted each model (with its default settings) to the training data, and calculated the recall score using the test set. Additionally, if the estimator had an `n_jobs` parameter, we set it to `-1` to use all the available cores for computations. This way, we could speed up the model's training, provided our machine has multiple cores/threads available. The goal of this step was to investigate the performance of the individual base models so that we could compare them to the stacked ensemble.

In *Step 5*, we first defined the meta-model (logistic regression) and the 5-fold stratified cross-validation scheme. Then, we instantiated the `StackingClassifier` by providing the list of the base classifiers, together with the cross-validation scheme and the meta-model. In the scikit-learn implementation of stacking, the base learners are fitted using the entire training set. Then, in order to avoid overfitting and improve the model's generalization, the meta-estimator uses the selected cross-validation scheme to train the model on the out-samples. To be precise, it uses `cross_val_predict` for this task.



A possible shortcoming of this approach is that applying cross-validation only to the meta-learner can result in overfitting of the base learners. Different libraries (mentioned in the *There's more...* section) employ different approaches to cross-validation with stacked ensembles.

In the last step, we tried to improve the performance of the stacked ensemble by modifying its two characteristics. First, we changed the level 1 model from logistic regression to a Random Forest classifier. Second, we allowed the level 1 model to use the features used by the level 0 base models. To do so, we set the `passthrough` argument to `True` while instantiating the `StackingClassifier`.

There's more...

In order to get a better understanding of stacking, we can take a peek at the output of *Step 1*, which is the data being used to train the level 1 model. To get that data, we can use the `transform` method of a fitted `StackedClassifier`. Alternatively, we can use the familiar `fit_transform` method when the classifier was not fitted. In our case, we look into the stacked ensemble using both the predictions and original data as features:

```
level_0_names = [f"model[{i}]_pred" for i in range(len(base_models))]

level_0_df = pd.DataFrame(
    stack_clf.transform(X_train),
    columns=level_0_names + list(X.columns)
)

level_0_df.head()
```

Executing the snippet generates the following table (abbreviated):

	dec_tree_pred	log_reg_pred	svc_pred	naive_bayes_pred	V1	V2	V3
0	0.0	0.000067	-1.042724	3.814708e-18	0.862468	-0.582668	-0.800214
1	0.0	0.000197	-1.011043	4.686999e-17	0.902013	-0.081009	-1.688308
2	0.0	0.000105	-1.064856	1.901214e-08	-0.452072	0.383882	0.277401
3	0.0	0.000018	-1.064256	2.215227e-17	1.014098	-1.115745	-0.483481
4	0.0	0.000041	-1.061369	3.443033e-17	-0.209097	-0.767259	-0.033341

Figure 14.17: Preview of the input for the level 1 model in the stacking ensemble

We can see that the first four columns correspond to the predictions made by the base learners. Next to those, we can see the rest of the features, that is, those used by the base learners to generate their predictions.

It is also worth mentioning that when using the `StackingClassifier` we can use various outputs of the base models as inputs for the level 1 model. For example, we can either use the predicted probabilities/scores or the predicted labels. Using the default settings of the `stack_method` argument, the classifier will try to use the following types of outputs (in that specific order): `predict_proba`, `decision_function`, and `predict`.



If we had used `stack_method="predict"`, we would have seen four columns of zeros and ones corresponding to the models' class predictions (using the default decision threshold of 0.5).

In this recipe, we presented a simple example of a stacked ensemble. There are multiple ways in which we could try to further improve it. Some of the possible extensions include:

- Adding more layers to the stacked ensemble
- Using more diverse models, such as k-NN, boosted trees, neural networks, and so on
- Tuning the hyperparameters of the base classifiers and/or the meta-model



The `ensemble` module of `scikit-learn` also contains a `VotingClassifier`, which can aggregate the predictions of multiple classifiers. `VotingClassifier` uses one of the two available voting schemes. The first one is `hard`, and it is simply the majority vote. The `soft` voting scheme uses the `argmax` of the sums of the predicted probabilities to predict the class label.

There are also other libraries providing stacking functionalities:

- `vecstack`
- `mlxtend`
- `h2o`

These libraries also differ in the way they approach stacking, for example, how they split the data or how they handle potential overfitting and data leakage. Please refer to the respective documentation for more details.

See also

Additional resources are available here:

- Raschka, S. 2018. “MLxtend: Providing machine learning and data science utilities and extensions to Python’s scientific computing stack.” *The Journal of Open Source Software* 3(24): 638.
- Wolpert, D. H. 1992. “Stacked generalization”. *Neural networks* 5(2): 241–259.

Bayesian hyperparameter optimization

In the *Tuning hyperparameters using grid search and cross-validation* recipe in the previous chapter, we described how to use various flavors of grid search to find the best possible set of hyperparameters for our model. In this recipe, we introduce an alternative approach to finding the optimal set of hyperparameters, this time based on the Bayesian methodology.

The main motivation for the Bayesian approach is that both grid search and randomized search make uninformed choices, either through an exhaustive search over all combinations or through a random sample. This way, they spend a lot of time evaluating combinations that result in far from optimal performance, thus basically wasting time. That is why the Bayesian approach makes informed choices of the next set of hyperparameters to evaluate, this way reducing the time spent on finding the optimal set. One could say that the Bayesian methods try to limit the time spent evaluating the objective function by spending more time on selecting the hyperparameters to investigate, which in the end is computationally cheaper.

A formalization of the Bayesian approach is **Sequential Model-Based Optimization (SMBO)**. On a very high level, SMBO uses a surrogate model together with an acquisition function to iteratively (hence “sequential”) select the most promising hyperparameters in the search space in order to approximate the actual objective function.

In the context of Bayesian HPO, the true objective function is often the cross-validation error of a trained machine learning model. It can be computationally very expensive and can take hours (or even days) to calculate. That is why in SMBO we create a **surrogate model**, which is a probability model of the objective function built using its past evaluations. It maps the input values (hyperparameters) to a probability of a score on the true objective function. Hence, we can think of it as an approximation of the true objective function. In the approach we follow (the one used by the `hyperopt` library), the surrogate model is created using the **Tree-Structured Parzen Estimator (TPE)**. Other possibilities include Gaussian processes or Random Forest regression.

In each iteration, we first fit the surrogate model to all observations of the target function we made so far. Then, we apply the acquisition function (such as **Expected Improvement**) to determine the next set of hyperparameters based on their expected utility. Intuitively, this approach uses the history of past evaluations to make the best possible selection for the next iteration. Values close to the ones that performed well in the past are more likely to improve the overall performance than those that historically performed poorly. The acquisition function also defines a balance between the exploration of new areas in the hyperparameter space and the exploitation of the areas that are already known to provide favorable results.

The simplified steps of Bayesian optimization are:

1. Create the surrogate model of the true objective function.
2. Find a set of hyperparameters that performs best on the surrogate.
3. Use that set to evaluate the true objective function.
4. Update the surrogate, using the results from evaluating the true objective.
5. Repeat Steps 2–4, until reaching the stop criterion (the specified maximum number of iterations or amount of time).

From these steps, we see that the longer the algorithm runs, the closer the surrogate function approximates the true objective function. That is because with each iteration it is updated based on the evaluation of the true objective function, and thus with each run it is a bit “less wrong.”

As we have already mentioned, the biggest advantage of Bayesian HPO is that it decreases the time spent searching for the optimal set of parameters. That is especially significant when the number of parameters is high and evaluating the true objective is computationally expensive. However, it also comes with a few possible shortcomings:

- Some steps of the SMBO procedure cannot be executed in parallel, as the algorithm selects the set of hyperparameters sequentially based on past results.
- Choosing a proper distribution/scale for the hyperparameters can be tricky.
- Exploration versus exploitation bias—when the algorithm finds a local optimum, it might concentrate on hyperparameter values around it, instead of exploring potential new values located far away in the search space. Randomized search is not troubled by this issue, as it does not concentrate on any values.
- The values of hyperparameters are selected independently. For example, in Gradient Boosted Trees, it is recommended to jointly consider the learning rate and the number of estimators, in order to avoid overfitting and reduce computation time. TPE would not be able to discover this relationship. In cases where we know about such a relation, we can partially overcome this problem by using different choices to define the search space.



In this brief introduction, we presented a high-level overview of the methodology. However, there is much more ground to cover in terms of surrogate models, acquisition functions, and so on. That is why we refer to a list of papers in the *See also* section for a more in-depth explanation.

In this recipe, we use the Bayesian hyperparameter optimization to tune a LightGBM model. We chose this model as it provides a very good balance between performance and training time. We will be using the already familiar credit card fraud dataset, which is a highly imbalanced dataset.

How to do it...

Execute the following steps to run Bayesian hyperparameter optimization of a LightGBM model:

1. Load the libraries:

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.model_selection import (cross_val_score,
                                      StratifiedKFold)
from lightgbm import LGBMClassifier

from hyperopt import hp, fmin, tpe, STATUS_OK, Trials, space_eval
from hyperopt.pyll import scope
from hyperopt.pyll.stochastic import sample

from chapter_14_utils import performance_evaluation_report
```

2. Define parameters for later use:

```
N_FOLDS = 5
MAX_EVALS = 200
RANDOM_STATE = 42
EVAL_METRIC = "recall"
```

3. Load and prepare the data:

```
df = pd.read_csv("../Datasets/credit_card_fraud.csv")

X = df.copy().drop(columns=["Time"])
y = X.pop("Class")

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=RANDOM_STATE
)
```

4. Train the benchmark LightGBM model with the default hyperparameters:

```
clf = LGBMClassifier(random_state=RANDOM_STATE)
clf.fit(X_train, y_train)

benchmark_perf = performance_evaluation_report(
    clf, X_test, y_test,
    show_plot=True,
    show_pr_curve=True
)
print(f'Recall: {benchmark_perf["recall"]:.4f}')
```

Executing the snippet generates the following plot:

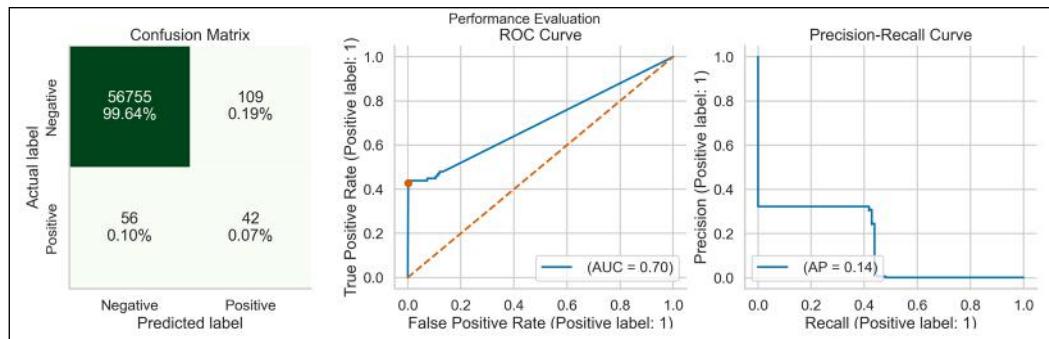


Figure 14.18: Performance evaluation of the benchmark LightGBM model

Additionally, we learned that the benchmark's recall score on the test set is equal to 0.4286.

5. Define the objective function:

```
def objective(params, n_folds=N_FOLDS,
             random_state=RANDOM_STATE,
             metric=EVAL_METRIC):

    model = LGBMClassifier(**params, random_state=random_state)
    k_fold = StratifiedKFold(n_folds, shuffle=True,
                            random_state=random_state)

    scores = cross_val_score(model, X_train, y_train,
                            cv=k_fold, scoring=metric)
    loss = -1 * scores.mean()

    return {"loss": loss, "params": params, "status": STATUS_OK}
```

6. Define the search space:

```
search_space = {
    "n_estimators": hp.choice("n_estimators", [50, 100, 250, 500]),
    "boosting_type": hp.choice(
        "boosting_type", ["gbdt", "dart", "goss"]
    ),
    "is_unbalance": hp.choice("is_unbalance", [True, False]),
    "max_depth": scope.int(hp.uniform("max_depth", 3, 20)),
    "num_leaves": scope.int(hp.quniform("num_leaves", 5, 100, 1)),
    "min_child_samples": scope.int(
        hp.quniform("min_child_samples", 20, 500, 5)
    ),
    "colsample_bytree": hp.uniform("colsample_bytree", 0.3, 1.0),
    "learning_rate": hp.loguniform(
        "learning_rate", np.log(0.01), np.log(0.5)
    ),
    "reg_alpha": hp.uniform("reg_alpha", 0.0, 1.0),
    "reg_lambda": hp.uniform("reg_lambda", 0.0, 1.0),
}
```

We can generate a single draw from the sample space using the `sample` function:

```
sample(search_space)
```

Executing the snippet prints the following dictionary:

```
{'boosting_type': 'gbdt',
 'colsample_bytree': 0.5718346953027432,
 'is_unbalance': False,
 'learning_rate': 0.44862566076557925,
 'max_depth': 3,
 'min_child_samples': 75,
 'n_estimators': 250,
 'num_leaves': 96,
 'reg_alpha': 0.31830737977056545,
 'reg_lambda': 0.637449220342909}
```

7. Find the best hyperparameters using Bayesian HPO:

```
trials = Trials()
best_set = fmin(fn=objective,
                 space=search_space,
                 algo=tpe.suggest,
                 max_evals=MAX_EVALS,
                 trials=trials,
                 rstate=np.random.default_rng(RANDOM_STATE))
```

8. Inspect the best set of hyperparameters:

```
space_eval(search_space , best_set)
```

Executing the snippet prints the list of the best hyperparameters:

```
{'boosting_type': 'dart',
 'colsample_bytree': 0.8764301395665521,
 'is_unbalance': True,
 'learning_rate': 0.019245717855584647,
 'max_depth': 19,
 'min_child_samples': 160,
 'n_estimators': 50,
 'num_leaves': 16,
 'reg_alpha': 0.3902317904740905,
 'reg_lambda': 0.48349252432635764}
```

9. Fit a new model using the best hyperparameters:

```
tuned_lgbm = LGBMClassifier(
    **space_eval(search_space, best_set),
    random_state=RANDOM_STATE
)
tuned_lgbm.fit(X_train, y_train)
```

10. Evaluate the fitted model on the test set:

```
tuned_perf = performance_evaluation_report(
    tuned_lgbm, X_test, y_test,
    show_plot=True,
    show_pr_curve=True
)

print(f'Recall: {tuned_perf["recall"]:.4f}')
```

Executing the snippet generates the following plot:

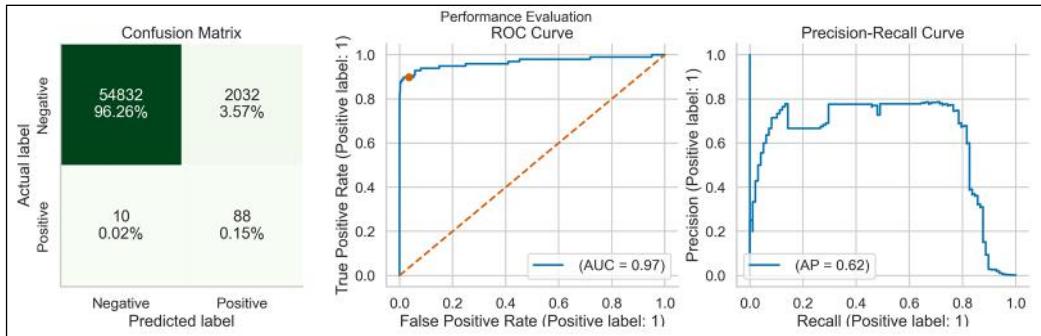


Figure 14.19: Performance evaluation of the tuned LightGBM model

We can see that the tuned model achieved better performance on the test set. To make it more concrete, its recall score was **0.8980**, as compared to the benchmark value of **0.4286**.

How it works...

After loading the required libraries, we defined a set of parameters that we used in this recipe: the number of folds for cross-validation, the maximum number of iterations in the optimization procedure, the random state, and the metric used for optimization.

In *Step 3*, we imported the dataset and created the training and test sets. We described a few preprocessing steps in previous recipes, so please refer to those for more information. Then, we trained a benchmark LightGBM model using the default hyperparameters.



While using LightGBM, we can actually define a few random seeds. There are separate ones used for bagging and selecting a subset of features for each tree. Also, there is a `deterministic` flag that we can specify. To make the results fully reproducible, we should also make sure those additional settings are correctly specified.

In *Step 5*, we defined the true objective function (the one for which the Bayesian optimization will create a surrogate). The function takes the set of hyperparameters as inputs and uses stratified 5-fold cross-validation to calculate the loss value to be minimized. In the case of fraud detection, we want to detect as much fraud as possible, even if it means creating more false positives. That is why we selected recall as the metric of interest. As the optimizer will minimize the function, we multiplied it by -1 to create a maximization problem. The function must return either a single value (the loss) or a dictionary with at least two key-value pairs:

- `loss`—The value of the true objective function.
- `status`—An indicator that the loss value was calculated correctly. It can be either `STATUS_OK` or `STATUS_FAIL`.

Additionally, we returned the set of hyperparameters used for evaluating the objective function. We will get back to it in the *There's more...* section.



We used the `cross_val_score` function to calculate the validation score. However, there are cases in which we might want to manually iterate over the folds created with `StratifiedKFold`. One such case would be to access more functionalities of the native API of LightGBM, for example, early stopping.

In *Step 6*, we defined the hyperparameter grid. The search space is defined as a dictionary, but in comparison to the spaces defined for `GridSearchCV`, we used `hyperopt`'s built-in functions, such as the following:

- `hp.choice(label, list)`—returns one of the indicated options.
- `hp.uniform(label, lower_value, upper_value)`—the uniform distribution between two values.
- `hp.quniform(label, low, high, q)`—the quantized (or discrete) uniform distribution between two values. In practice, it means that we obtain uniformly distributed, evenly spaced (determined by `q`) integers.
- `hp.loguniform(label, low, high)`—the logarithm of the returned value is uniformly distributed. In other words, the returned numbers are evenly distributed on a logarithmic scale. Such a distribution is useful for exploring values that vary over several orders of magnitude. For example, when tuning the learning rate we would like to test values such as 0.001, 0.01, 0.1, and 1, instead of a uniformly distributed set between 0 and 1.
- `hp.randint(label, upper_value)`—returns a random integer in the range $[0, \text{upper_value}]$.

Bear in mind that in this setup we had to define the names (denoted as `label` in the snippets above) of the hyperparameters twice. Additionally, in some cases, we wanted to force the values to be integers using `scope.int`.

In *Step 7*, we ran the Bayesian optimization to find the best set of hyperparameters. First, we defined the `Trials` object, which was used for storing the history of the search. We could even use it to resume a search or expand an already finished one, that is, increase the number of iterations using the already stored history.

Second, we ran the optimization by passing the objective function, the search space, the surrogate model, the maximum number of iterations, and the `trials` object for storing the history. For more details on tuning the TPE algorithm, please refer to `hyperopt`'s documentation. Additionally, we set the value of `rstate`, which is `hyperopt`'s equivalent of `random_state`. We can easily store the `trials` object in a pickle file for later use. To do so, we can use the `pickle.dump` and `pickle.load` functions.



After running the Bayesian HPO, the `trials` object contains a lot of interesting and useful information. We can find the best set of hyperparameters under `trials.best_trial`, while `trials.results` contains all the explored sets of hyperparameters. We will be using this information in the *There's more...* section.

In Step 8, we inspected the best set of hyperparameters. Instead of just printing the dictionary, we had to use the `space_eval` function. This is because just by printing the dictionary we will see the indices of any categorical features instead of their names. As an example, by printing the `best_set` dictionary we could potentially see a 0 instead of 'gbdt' for the `boosting_type` hyperparameter.

In the last two steps, we trained a LightGBM classifier using the identified hyperparameters and evaluated its performance on the test set.

There's more...

There are still quite a lot of interesting and useful things to mention about Bayesian hyperparameter optimization. We try to present those in the following subsections. For brevity's sake, we do not present all the code here. For the complete code walk-through, please refer to the Jupyter notebook available in the book's GitHub repository.

Conditional hyperparameter spaces

Conditional hyperparameter spaces can be useful when we would like to experiment with different machine learning models, each of those coming with completely separate hyperparameters. Alternatively, some hyperparameters are simply not compatible with others, and this should be accounted for while tuning the model.

In the case of LightGBM, an example could be the following pair: `boosting_type` and `subsample`/`subsample_freq`. The boosting type "goss" is not compatible with subsampling, that is, selecting only a subsample of the training observations for each iteration. That is why we would like to set `subsample` to 1 when we are using GOSS, but tune it otherwise. `subsample_freq` is a complementary hyperparameter that determines how often (every n -th iteration) we should use subsampling.

We define a conditional search space using `hp.choice` in the following snippet:

```
conditional_search_space = {
    "boosting_type": hp.choice("boosting_type", [
        {"boosting_type": "gbdt",
         "subsample": hp.uniform("gdbt_subsample", 0.5, 1),
         "subsample_freq": scope.int(
             hp.uniform("gdbt_subsample_freq", 1, 20)
         )},
        {"boosting_type": "dart",
         "subsample": hp.uniform("dart_subsample", 0.5, 1),
         "subsample_freq": scope.int(
             hp.uniform("dart_subsample_freq", 1, 20)
         )},
    ])
}
```

```

        {"boosting_type": "goss",
         "subsample": 1.0,
         "subsample_freq": 0},
      ]),
      "n_estimators": hp.choice("n_estimators", [50, 100, 250, 500]),
    }
}

```

And an example of a draw from this space looks as follows:

```

{'boosting_type': {'boosting_type': 'dart',
                   'subsample': 0.9301284507624732,
                   'subsample_freq': 17},
 'n_estimators': 250}

```

There is one more step that we need to take before being able to use such a draw for our Bayesian HPO. As the search space is initially nested, we have to assign the drawn samples to the top-level key in the dictionary. We do so in the following snippet:

```

# draw from the search space
params = sample(conditional_search_space)

# retrieve the conditional parameters, set to default if missing
subsample = params["boosting_type"].get("subsample", 1.0)
subsample_freq = params["boosting_type"].get("subsample_freq", 0)

# fill in the params dict with the conditional values
params["boosting_type"] = params["boosting_type"]["boosting_type"]
params["subsample"] = subsample
params["subsample_freq"] = subsample_freq

params

```

The `get` method extracts the value of the requested key from the dictionary or returns the default value if the requested key does not exist.

Executing the snippet returns a properly formatted dictionary:

```

{'boosting_type': 'dart',
 'n_estimators': 250,
 'subsample': 0.9301284507624732,
 'subsample_freq': 17}

```

Lastly, we should place the code cleaning up the dictionary in the objective function, which we then pass to the optimization routine.

In the Jupyter notebook, we have also tuned the LightGBM with the conditional search space. It achieved a recall score of **0.8980** on the test set, which is the same score as the model tuned without the conditional search space.

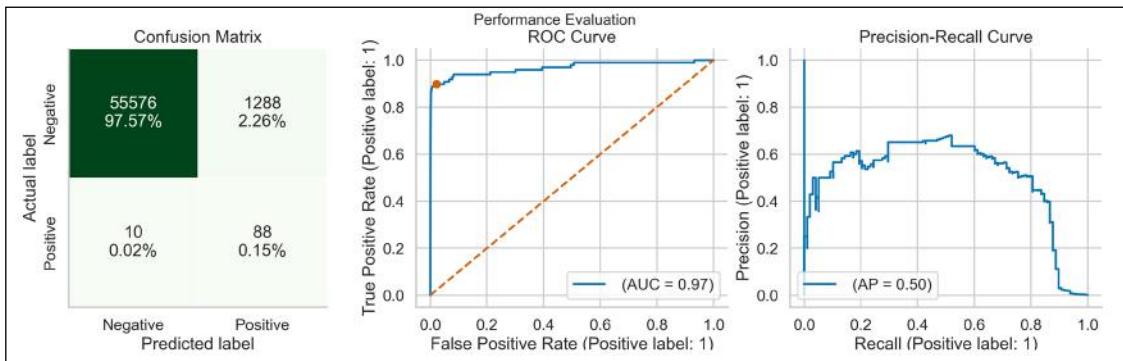


Figure 14.20: Performance evaluation of the LightGBM model tuned with the conditional search space

A deep dive into the explored hyperparameters

We have mentioned that hyperopt offers a wide range of distributions from which we could sample. It will be much easier to understand when we actually see what the distributions look like. First, we inspect the distribution of the learning rate. We have specified it as:

```
hp.loguniform("learning_rate", np.log(0.01), np.log(0.5))
```

In the following figure, we can see a **kernel density estimate (KDE)** plot of 10,000 random draws from the log-uniform distribution of the learning rate.

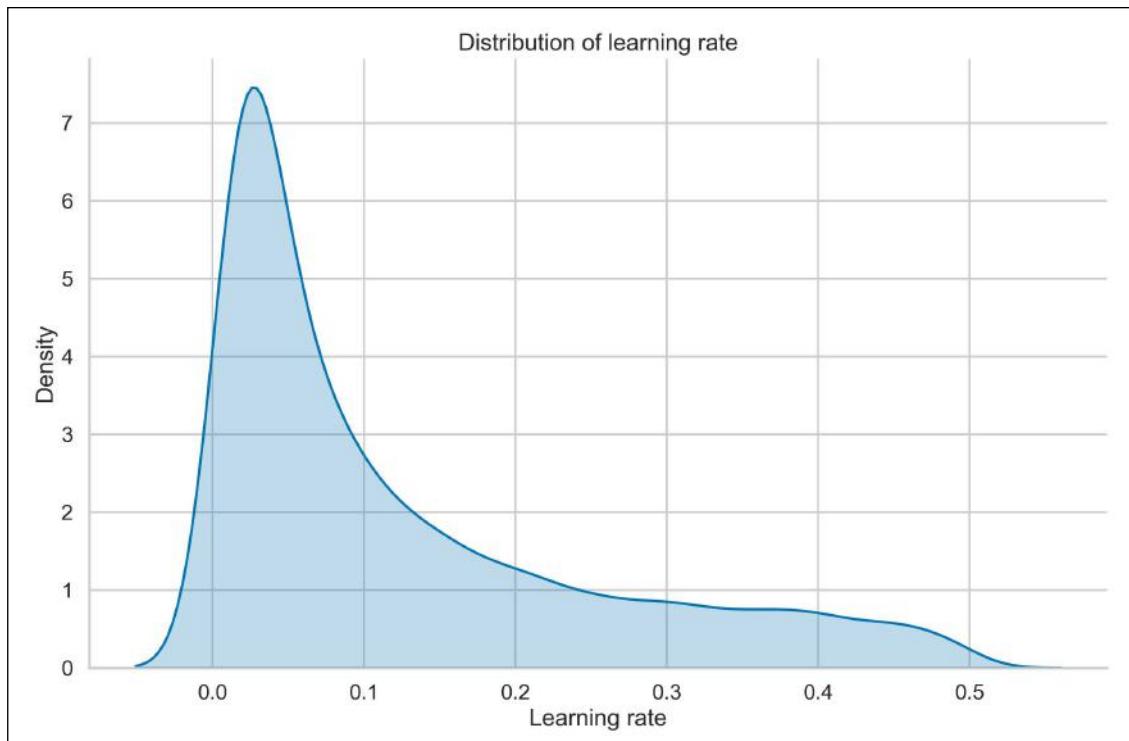


Figure 14.21: Distribution of the learning rate

As intended, we can see that the distribution puts more weight on observations from several orders of magnitude.

The next distribution worth inspecting is the quantized uniform distribution that we have used for the `min_child_samples` hyperparameter. We defined it as:

```
scope.int(hp.quniform("min_child_samples", 20, 500, 5))
```

In the following figure, we can see that the distribution reflects the assumptions we set for it, that is, the evenly spaced integers are uniformly distributed. In our case, we sampled every fifth integer. To keep the plot readable, we only displayed the first 20 bars. But the full distribution goes to 500, just as we have specified.

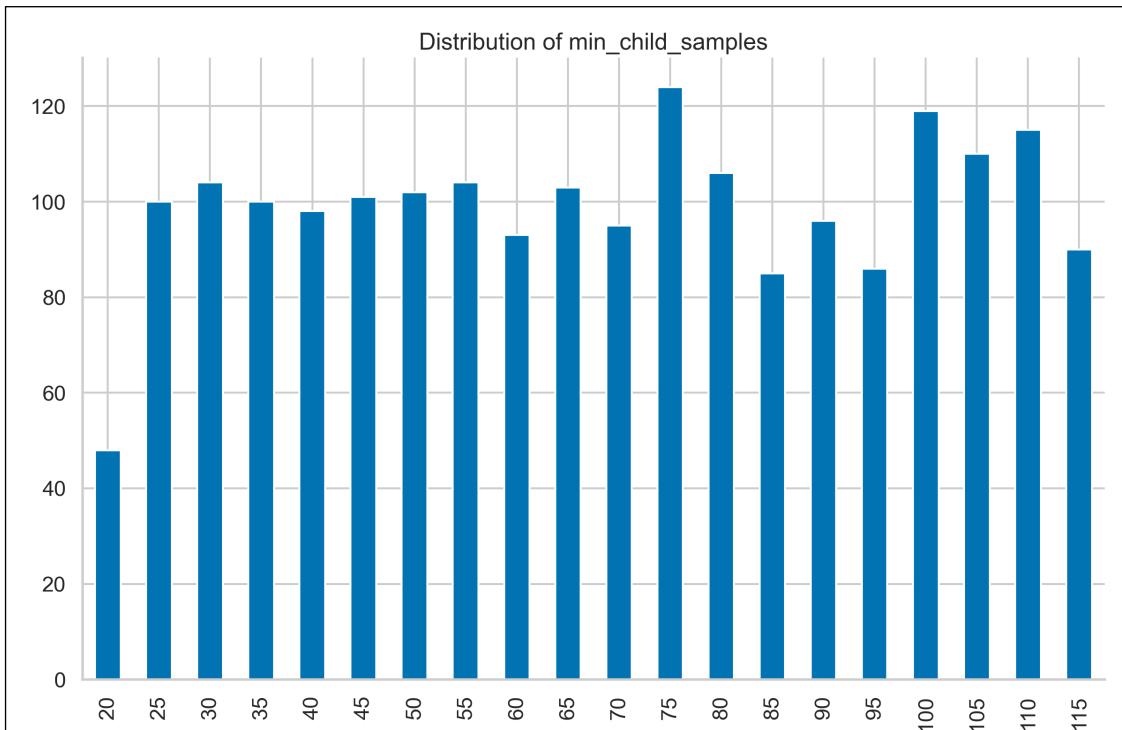


Figure 14.22: Distribution of the `min_child_samples` hyperparameter

So far, we have only looked at the information available in the search space. However, we can also derive much more information from the `Trials` object, which stores the entire history of the Bayesian HPO procedure, that is, which hyperparameters were explored and what the resulting score was.

For this part, we use the `Trials` object containing the search history, using the search space without the conditional `boosting_type` tuning. In order to easily explore that data, we prepare a `DataFrame` containing the required information per iteration: the hyperparameters and the value of the loss function. We can extract the information from `trials.results`. This is the reason why we additionally passed the `params` object to the final dictionary while defining the `objective` function.

Initially, the hyperparameters are stored in one column as a dictionary. We can use the `json_normalize` function to break them up into separate columns:

```
from pandas.io.json import json_normalize
results_df = pd.DataFrame(trials.results)
params_df = json_normalize(results_df["params"])
results_df = pd.concat([results_df.drop("params", axis=1), params_df],
                      axis=1)
results_df["iteration"] = np.arange(len(results_df)) + 1
results_df.sort_values("loss")
```

Executing the snippet prints the following table:

	loss	status	boosting_type	colsample_bytree	is_unbalance	learning_rate	max_depth	min_child_samples	n_estimators	num_leaves	reg_alpha	reg_lambda	iteration
150	-0.901071	ok	dart	0.876430	True	0.019246	19	160	50	16	0.390232	0.483493	151
168	-0.900941	ok	dart	0.946974	True	0.020825	18	235	50	8	0.366738	0.320576	169
167	-0.898475	ok	dart	0.951135	True	0.024895	18	135	50	8	0.362945	0.418759	168
34	-0.896040	ok	goss	0.307030	True	0.086375	3	415	50	82	0.236233	0.408269	35
155	-0.896008	ok	dart	0.901467	True	0.022619	17	135	50	5	0.320720	0.482145	156
...
6	-0.639695	ok	gbdt	0.393251	False	0.018363	19	265	50	13	0.996987	0.699079	7
52	-0.416326	ok	dart	0.841847	False	0.012526	6	320	100	73	0.072400	0.769000	53
164	-0.258845	ok	dart	0.921732	False	0.014171	18	110	50	25	0.049658	0.293709	165
199	-0.088608	ok	dart	0.993957	False	0.016164	15	210	50	54	0.242146	0.560007	200
181	-0.000000	ok	dart	0.837517	False	0.012799	15	160	50	23	0.428376	0.251945	182

Figure 14.23: A snippet of the DataFrame containing all the explored hyperparameter combinations and their corresponding losses

For brevity, we only printed a few of the available columns. Using this information, we can further explore the optimization that resulted in the best set of hyperparameters. For example, we can see that the best score was achieved in the 151st iteration (the first row of the DataFrame has an index of 150 and indices in Python start with 0).

In the next figure, we have plotted the two distributions of the `colsample_bytree` hyperparameter: the one we defined as the prior for sampling, and the one that was actually sampled during the Bayesian optimization. Additionally, we plotted the evolution of the hyperparameter over iterations and added a regression line to indicate the direction of change.

In the left plot, we can see that the posterior distribution of `colsample_bytree` was concentrated toward the right side, indicating the higher range of considered values. By inspecting the KDE plots it seems that there is a non-zero density for values above 1, which should not be allowed.

This is just the artifact from using the plotting method; in the `Trials` object we can confirm that not a single value above 1.0 was sampled during the optimization. In the right plot, the values of `colsample_bytree` seem to be scattered all over the allowed range. By looking at the regression line, it seems that there is a somewhat increasing trend.

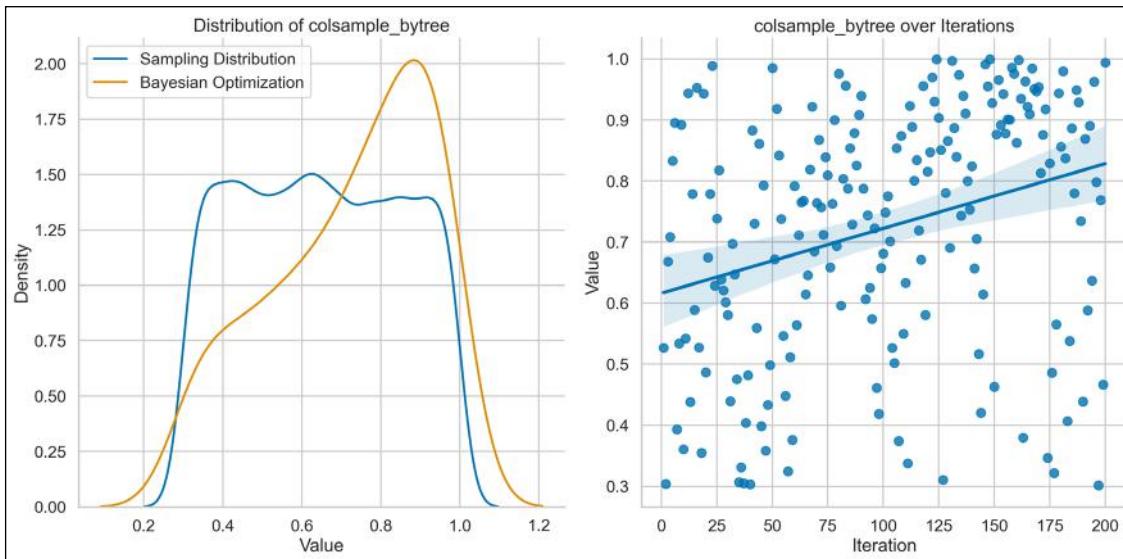


Figure 14.24: Distribution of the `colsample_bytree` hyperparameter

Lastly, we can look at the evolution of the loss over iterations. The loss represents the negative of the average recall score (from a 5-fold cross-validation on the training set). The lowest value (corresponding to maximum average recall) of -0.90 occurred in the 151st iteration. With a few exceptions, the loss is quite stable in the -0.75 to -0.85 range.

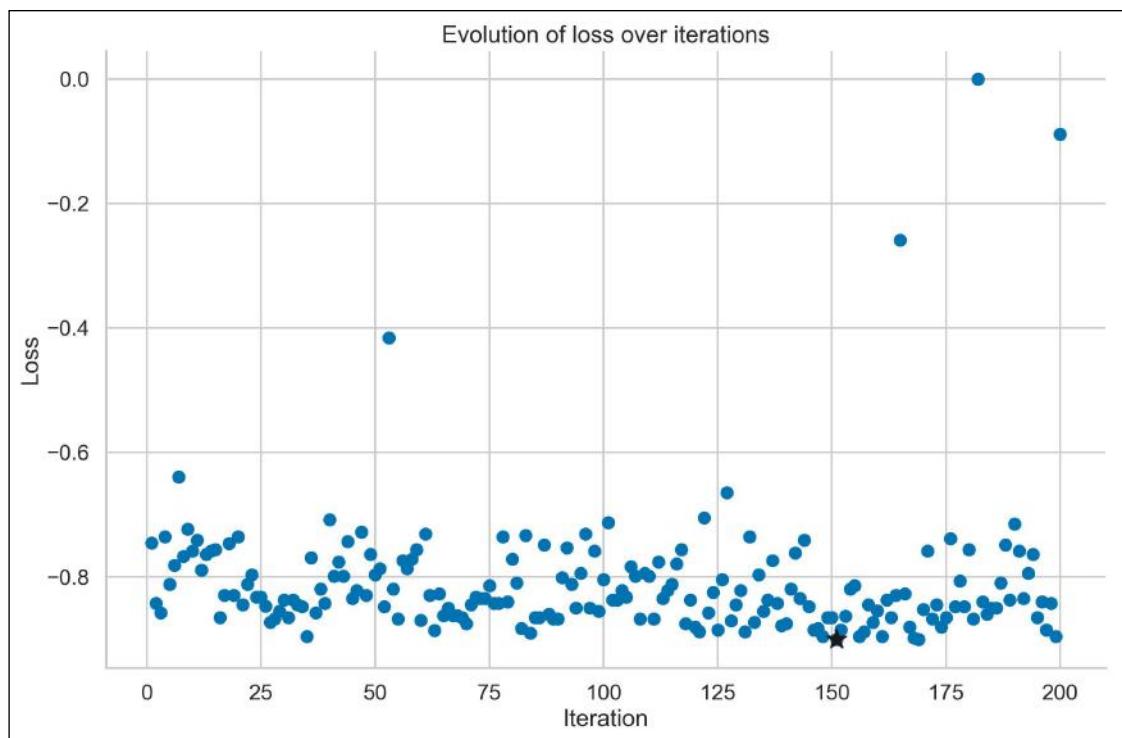


Figure 14.25: The evolution of the loss (average recall) over iterations. The best iteration is marked with a star

Other popular libraries for hyperparameter optimization

hyperopt is one of the most popular Python libraries for hyperparameter optimization. However, it is definitely not the only one. Below you can find a list of popular alternatives:

- optuna—a library offering vast hyperparameter tuning capabilities, including exhaustive Grid Search, Random Search, Bayesian HPO, and evolutionary algorithms.
- scikit-optimize—a library offering the BayesSearchCV class, which is a Bayesian drop-in replacement for scikit-learn's GridSearchCV.

- `hyperopt-sklearn`—a spin-off library of `hyperopt` offering model selection among machine learning algorithms from `scikit-learn`. It allows you to search for the best option among pre-processing steps and ML models, thus covering the entire scope of ML pipelines. The library covers almost all classifiers/regressors/preprocessing transformers available in `scikit-learn`.
- `ray[tune]`—Ray is an open-source, general-purpose distributed computing framework. We can use its `tune` module to run distributed hyperparameter tuning. It is also possible to combine `tune`'s distributed computing capabilities with other well-established libraries such as `hyperopt` or `optuna`.
- `Tpot`—TPOT is an AutoML tool that optimizes ML pipelines using genetic programming.
- `bayesian-optimization`—a library offering general-purpose Bayesian global optimization with Gaussian processes.
- `smac`—SMAC is a general tool for optimizing the parameters of arbitrary algorithms, including hyperparameter optimization of ML models.

See also

Additional resources are available here:

- Bergstra, J. S., Bardenet, R., Bengio, Y., & Kégl, B. 2011. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*: 2546–2554.
- Bergstra, J., Yamins, D., & Cox, D. D. 2013, June. Hyperopt: A Python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference*: 13–20.
- Bergstra, J., Yamins, D., Cox, D. D. 2013. Making a Science of Model Search: Hyperparameter Optimization in *Hundreds of Dimensions for Vision Architectures*. *Proc. of the 30th International Conference on Machine Learning* (ICML 2013).
- Claesen, M., & De Moor, B. 2015. “Hyperparameter search in machine learning.” *arXiv preprint arXiv:1502.02127*.
- Falkner, S., Klein, A., & Hutter, F. 2018, July. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*: 1437–1446. PMLR.
- Hutter, F., Kotthoff, L., & Vanschoren, J. 2019. *Automated machine learning: methods, systems, challenges*: 219. Springer Nature.
- Klein, A., Falkner, S., Bartels, S., Hennig, P., & Hutter, F. 2017, April. Fast Bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial intelligence and statistics*: 528–536. PMLR.
- Komor B., Bergstra J., & Eliasmith C. 2014. “Hyperopt-Sklearn: automatic hyperparameter configuration for Scikit-learn” *Proc. SciPy*.
- Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Hardt, M., Recht, B., & Talwalkar, A. 2018. Massively parallel hyperparameter tuning: <https://doi.org/10.48550/arXiv.1810.05934>
- Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & De Freitas, N. 2015. *Taking the human out of the loop: A review of Bayesian optimization*. *Proceedings of the IEEE*, 104(1): 148–175.

- Snoek, J., Larochelle, H., & Adams, R. P. 2012. Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*: 25.

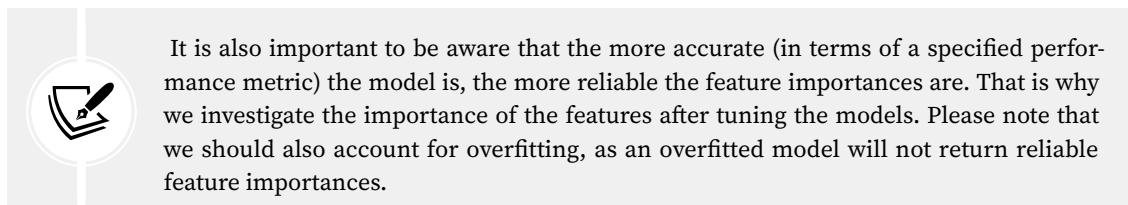
Investigating feature importance

We have already spent quite some time creating the entire pipeline and tuning the models to achieve better performance. However, what is equally—or in some cases even more—important is the model’s interpretability. That means not only giving an accurate prediction but also being able to explain the why behind it. For example, we can look into the case of customer churn. Knowing what the actual predictors of the customers leaving are might be helpful in improving the overall service and potentially making them stay longer.

In a financial setting, banks often use machine learning in order to predict a customer’s ability to repay credit or a loan. In many cases, they are obliged to justify their reasoning, that is, if they decline a credit application, they need to know exactly why this customer’s application was not approved. In the case of very complicated models, this might be hard, or even impossible.

We can benefit in multiple ways by knowing the importance of our features:

- By understanding the model’s logic, we can theoretically verify its correctness (if a sensible feature is a good predictor), but also try to improve the model by focusing only on the important variables.
- We can use the feature importances to only keep the x most important features (contributing to a specified percentage of total importance), which can not only lead to better performance by removing potential noise but also to a shorter training time.
- In some real-life cases, it makes sense to sacrifice some accuracy (or any other performance metric) for the sake of interpretability.



It is also important to be aware that the more accurate (in terms of a specified performance metric) the model is, the more reliable the feature importances are. That is why we investigate the importance of the features after tuning the models. Please note that we should also account for overfitting, as an overfitted model will not return reliable feature importances.

In this recipe, we show how to calculate the feature importance on an example of a Random Forest classifier. However, most of the methods are model-agnostic. In other cases, there are often equivalent approaches (such as in the case of XGBoost and LightGBM). We mention some of those in the *There’s more...* section. We briefly present the three selected methods of calculating feature importance.

Mean Decrease in Impurity (MDI): The default feature importance used by Random Forest (in `scikit-learn`), also known as the Gini importance. As we know, decision trees use a metric of impurity (Gini index/entropy/MSE) to create the best splits while growing. When training a decision tree, we can compute how much each feature contributes to decreasing the weighted impurity. To calculate the feature importance for the entire forest, the algorithm averages the decrease in impurity over all the trees.



While working with impurity-based metrics, we should focus on the ranking of the variables (relative values) rather than the absolute values of the feature importances (which are also normalized to add up to 1).

Here are the advantages of this approach:

- Fast calculation
- Easy to retrieve

Here are the disadvantages of this approach:

- Biased—It tends to inflate the importance of continuous (numerical) features or high-cardinality categorical variables. This can sometimes lead to absurd cases, whereby an additional random variable (unrelated to the problem at hand) scores high in the feature importance ranking.
- Impurity-based importances are calculated on the basis of the training set and do not reflect the model's ability to generalize to unseen data.

Drop-column feature importance: The idea behind this approach is very simple. We compare a model with all the features to a model with one of the features dropped for training and inference. We repeat this process for all the features.

Here is the advantage of this approach:

- Often considered the most accurate/reliable measure of feature importance

Here is the disadvantage of this approach:

- Potentially highest computation cost caused by retraining the model for each variant of the dataset

Permutation feature importance: This approach directly measures feature importance by observing how random reshuffling of each predictor influences the model's performance. The permutation procedure breaks the relationship between the feature and the target. Hence, the drop in the model's performance is indicative of how much the model is dependent on a particular feature. If the decrease in the performance after reshuffling a feature is small, then it was not a very important feature in the first place. Conversely, if the decrease in performance is significant, the feature can be considered an important one for the model.

The steps of the algorithm are:

1. Train the baseline model and record the score of interest.
2. Randomly permute (reshuffle) the values of one of the features, then use the entire dataset (with one reshuffled feature) to obtain predictions and record the score. The feature importance is the difference between the baseline score and the one from the permuted dataset.
3. Repeat the second step for all features.



For evaluating the performance, we can either use the training data or the validation/test set. Using one of the latter two has the additional benefit of gaining insights into the model's ability to generalize. For example, features that turn out to be important on the training set but not on the validation set might actually cause the model to overfit. For more discussion about the topic, please refer to the *Interpretable Machine Learning* book (referenced in the *See also* section).

Here are the advantages of this approach:

- Model-agnostic
- Reasonably efficient—no need to retrain the model at every step
- Reshuffling preserves the distribution of the variables

Here are the disadvantages of this approach:

- Computationally more expensive than the default feature importances
- Is likely to produce unreliable importances when features are highly correlated (see Strobl *et al.* for a detailed explanation)

In this recipe, we will explore the feature importance using the credit card default dataset we have already explored in the *Exploring ensemble classifiers* recipe.

Getting ready

For this recipe, we use the fitted Random Forest pipeline (called `rf_pipeline`) from the *Exploring ensemble classifiers* recipe. Please refer to this step in the Jupyter notebook to see all the initial steps not included here to avoid repetition.

How to do it...

Execute the following steps to evaluate the feature importance of a Random Forest model:

1. Import the libraries:

```
import numpy as np
import pandas as pd
from sklearn.inspection import permutation_importance
from sklearn.metrics import recall_score
from sklearn.base import clone
```

2. Extract the classifier and preprocessor from the fitted pipeline:

```
rf_classifier = rf_pipeline.named_steps["classifier"]
preprocessor = rf_pipeline.named_steps["preprocessor"]
```

3. Recover feature names from the preprocessing transformer and transform the training/test sets:

```
feat_names = list(preprocessor.get_feature_names_out())

X_train_preprocessed = pd.DataFrame(
    preprocessor.transform(X_train),
    columns=feat_names
)
X_test_preprocessed = pd.DataFrame(
    preprocessor.transform(X_test),
    columns=feat_names
)
```

4. Extract the MDI feature importance and calculate the cumulative importance:

```
rf_feat_imp = pd.DataFrame(rf_classifier.feature_importances_,
                            index=feat_names,
                            columns=[ "mdi" ])

rf_feat_imp[ "mdi_cumul" ] = np.cumsum(
    rf_feat_imp
    .sort_values("mdi", ascending=False)
    .loc[:, "mdi"]
).loc[feat_names]
```

5. Define a function for plotting the top x features in terms of their importance:

```
def plot_most_important_features(feat_imp, title,
                                  n_features=10,
                                  bottom=False):
    if bottom:
        indicator = "Bottom"
        feat_imp = feat_imp.sort_values(ascending=True)
    else:
        indicator = "Top"
        feat_imp = feat_imp.sort_values(ascending=False)

    ax = feat_imp.head(n_features).plot.barh()
    ax.invert_yaxis()
    ax.set(title=f"{title} ({indicator} {n_features})",
           xlabel="Importance",
           ylabel="Feature")

    return ax
```

We use the function as follows:

```
plot_most_important_features(rf_feat_imp["mdi"],  
                             title="MDI Importance")
```

Executing the snippet generates the following plot:

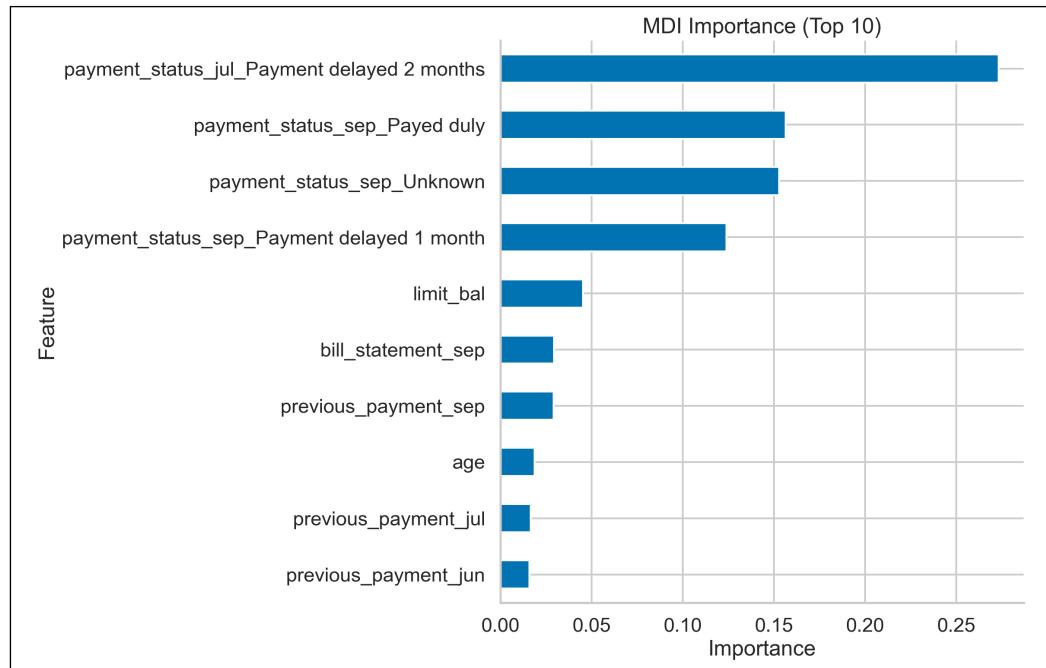


Figure 14.26: Top 10 most important features using the MDI metric

The most important features are categorical features indicating the payment status from July and September. After four of those, we can see continuous features such as `limit_balance`, `age`, various bill statements, and previous payments.

6. Plot the cumulative importance of the features:

```
x_values = range(len(feat_names))  
  
fig, ax = plt.subplots()  
ax.plot(x_values, rf_feat_imp["mdi_cumul"].sort_values(), "b-")  
ax.hlines(y=0.95, xmin=0, xmax=len(x_values),  
          color="g", linestyles="dashed")  
ax.set(title="Cumulative MDI Importance",  
      xlabel="# Features",  
      ylabel="Importance")
```

Executing the snippet generates the following plot:

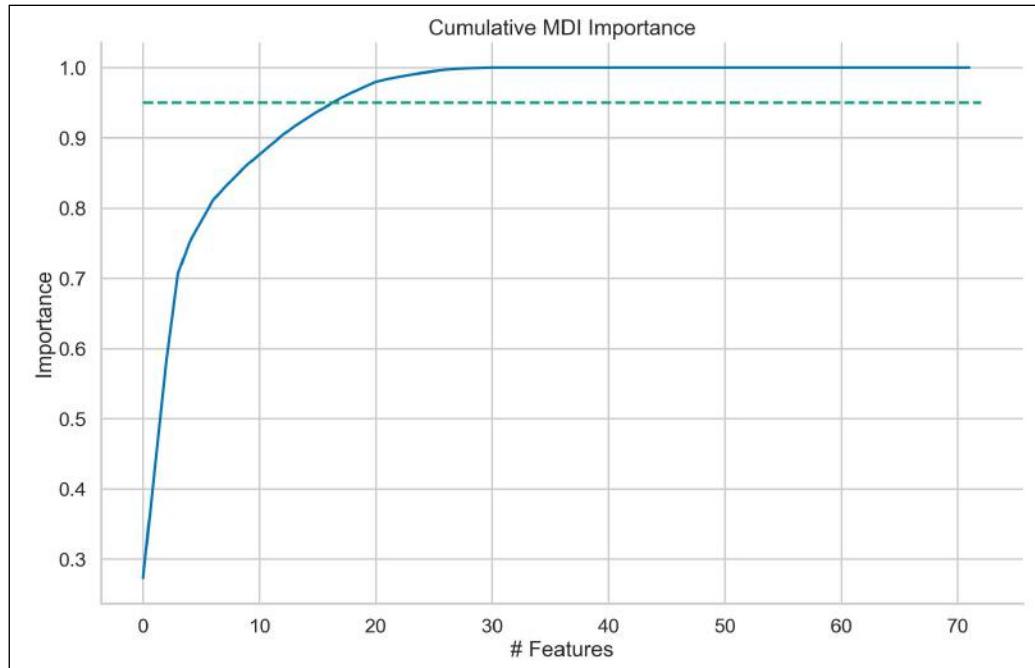


Figure 14.27: Cumulative MDI importance

The top 10 features account for 86.23% of the total importance, while the top 17 features account for 95% of the total importance.

7. Calculate and plot permutation importance using the training set:

```
perm_result_train = permutation_importance(  
    rf_classifier, X_train_preprocessed, y_train,  
    n_repeats=25, scoring="recall",  
    random_state=42, n_jobs=-1  
)  
  
rf_feat_imp["perm_imp_train"] = (  
    perm_result_train["importances_mean"]  
)  
  
plot_most_important_features(  
    rf_feat_imp["perm_imp_train"],  
    title="Permutation importance - training set"  
)
```

Executing the snippet generates the following plot:

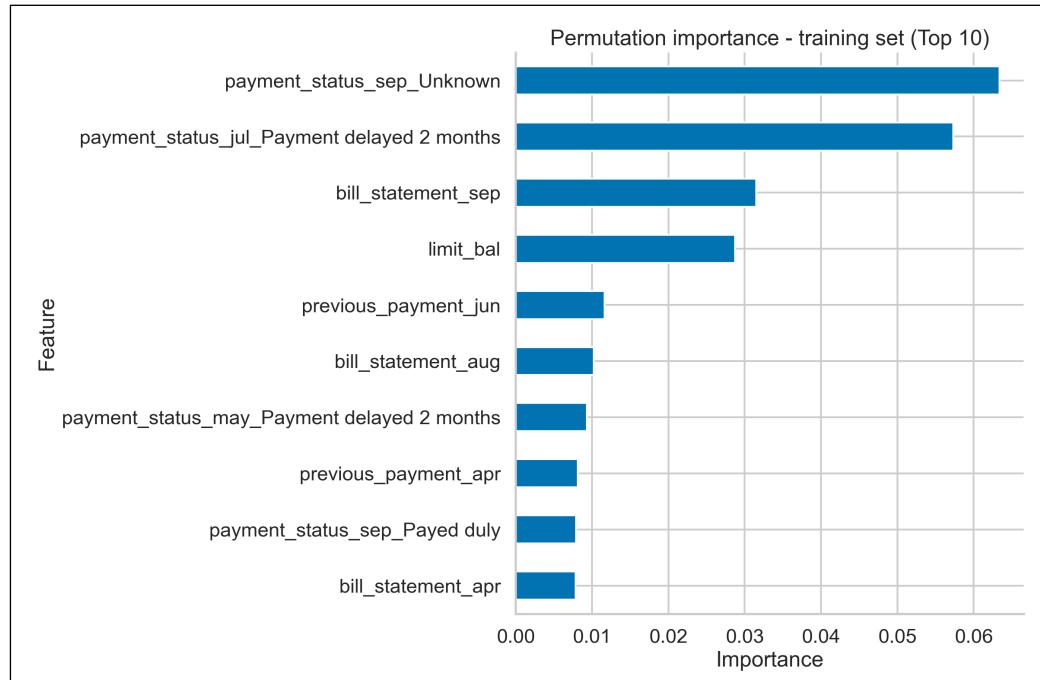


Figure 14.28: Top 10 most important features according to permutation importance calculated on the training set

We can see that the set of the most important features was reshuffled in comparison to the MDI importance. The most important now is `payment_status_sep_Unknown`, which is an undefined label (not assigned a clear meaning in the original paper) in the `payment_status_sep` categorical feature. We can also see that age is not among the top 10 most important features determined using this approach.

8. Calculate and plot permutation importance using the test set:

```
perm_result_test = permutation_importance(
    rf_classifier, X_test_preprocessed, y_test,
    n_repeats=25, scoring="recall",
    random_state=42, n_jobs=-1
)

rf_feat_imp["perm_imp_test"] = (
    perm_result_test["importances_mean"]
)
```

```
plot_most_important_features(  
    rf_feat_imp["perm_imp_test"],  
    title="Permutation importance - test set"  
)
```

Executing the snippet generates the following plot:

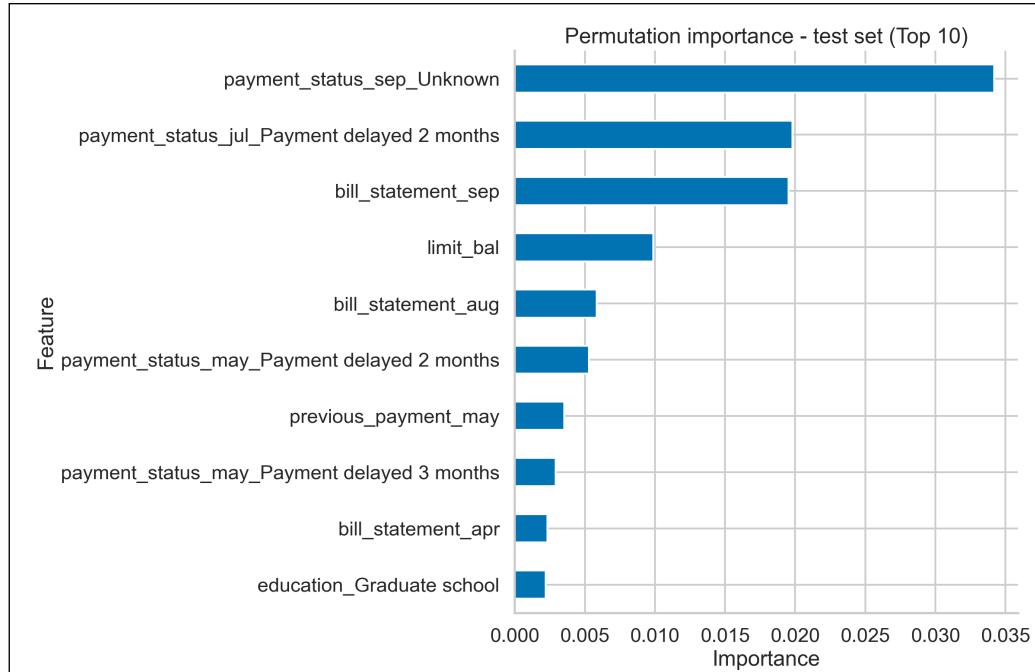


Figure 14.29: Top 10 most important features according to permutation importance calculated on the test set

Looking at the figures, we can state that the same four features were selected as the most important ones using the training and test sets. The other ones were slightly reshuffled.



If we notice that the feature importances calculated using the training and test sets are significantly different, we should investigate whether the model is overfitted. To solve that, we might want to apply some form of regularization. In this case, we could try increasing the value of the `min_samples_leaf` hyperparameter.

- Define a function for calculating the drop-column feature importance:

```
def drop_col_feat_imp(model, X, y, metric, random_state=42):
    model_clone = clone(model)
    model_clone.random_state = random_state
    model_clone.fit(X, y)
    benchmark_score = metric(y, model_clone.predict(X))

    importances = []

    for ind, col in enumerate(X.columns):
        print(f"Dropping {col} ({ind+1}/{len(X.columns)})")
        model_clone = clone(model)
        model_clone.random_state = random_state
        model_clone.fit(X.drop(col, axis=1), y)
        drop_col_score = metric(
            y, model_clone.predict(X.drop(col, axis=1)))
        importances.append(benchmark_score - drop_col_score)

    return importances
```

There are two things worth mentioning here:

- We fixed the `random_state`, as we are specifically interested in performance changes caused by removing a feature. Hence, we are controlling the source of variability during the estimation procedure.
- In this implementation, we use the training data for evaluation. We leave it as an exercise for the reader to modify the function to accept additional objects for evaluation.

- Calculate and plot the drop-column feature importance:

```
rf_feat_imp["drop_column_imp"] = drop_col_feat_imp(
    rf_classifier.set_params(**{"n_jobs": -1}),
    X_train_preprocessed,
    y_train,
    metric=recall_score,
    random_state=42
)
```

First, plot the top 10 most important features:

```
plot_most_important_features(  
    rf_feat_imp["drop_column_imp"],  
    title="Drop column importance"  
)
```

Executing the snippet generates the following plot:

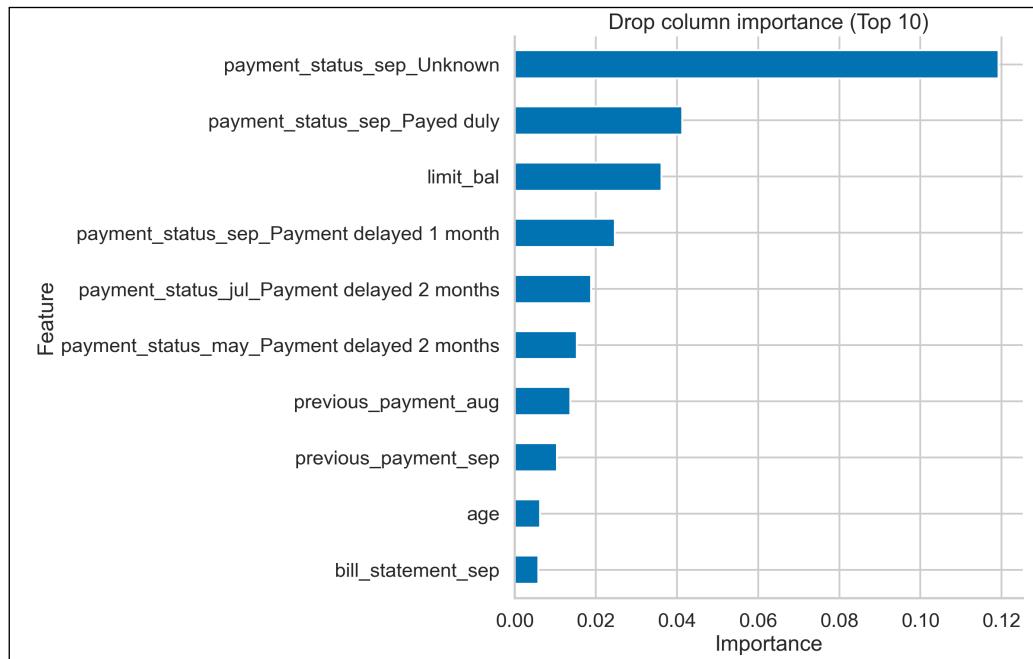


Figure 14.30: Top 10 most important features according to drop-column feature importance

Using the drop-column feature importance (evaluated on the training data), the most important feature was `payment_status_sep_Unknown`. The same feature was identified as the most important one using permutation feature importance.

Then, plot the 10 least important features:

```
plot_most_important_features(  
    rf_feat_imp["drop_column_imp"],  
    title="Drop column importance",  
    bottom=True  
)
```

Executing the snippet generates the following plot:

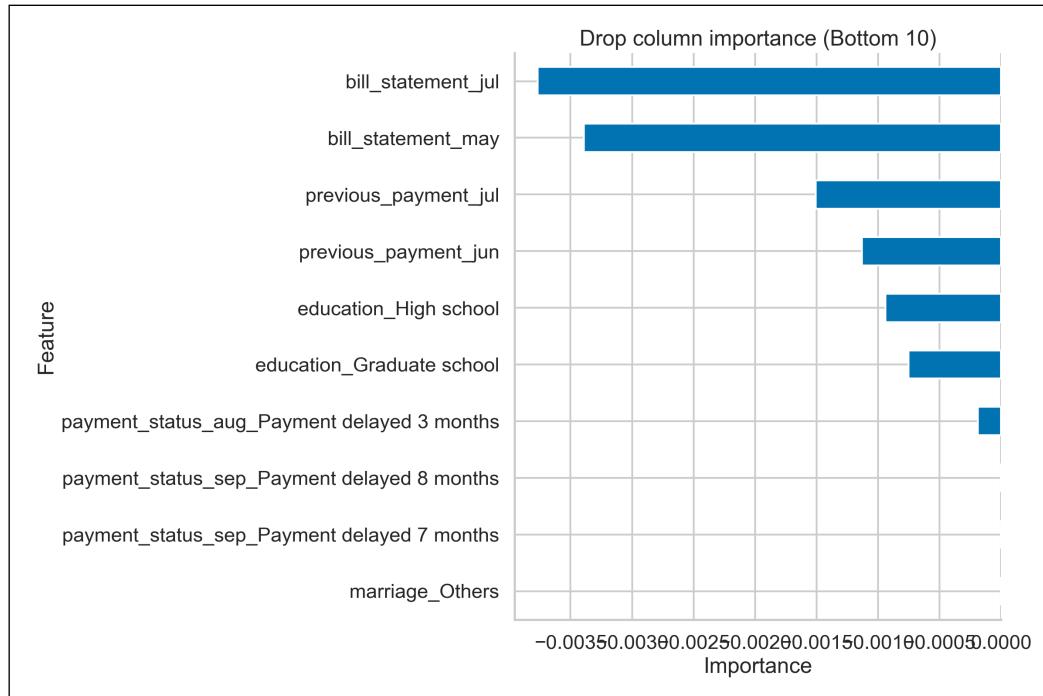


Figure 14.31: The 10 least important features according to drop-column feature importance

In the case of drop-column feature importance, negative importance indicates that removing a given feature from the model actually improves the performance. That is true as long as the considered metric treats higher values as better.

We can use these results to remove features that have negative importance and thus potentially improve the model's performance and/or reduce the training time.

How it works...

In *Step 1*, we imported the required libraries. Then, we extracted the classifier and the `ColumnTransformer` preprocessor from the pipeline. In this recipe, we worked with a tuned Random Forest classifier (using the hyperparameters determined in the *Exploring ensemble classifiers* recipe).

In *Step 3*, we first extracted the column names from the preprocessor using the `get_feature_names_out` method. Then, we prepared the training and test sets by applying the preprocessor's transformations.

In *Step 4*, we extracted the MDI feature importances using the `feature_importances_` attribute of the fitted Random Forest classifier. The values were automatically normalized so that they added up to 1. Additionally, we calculated the cumulative feature importance.

In *Step 5*, we defined a helper function to plot the most/least important features and plotted the top 10 most important features, calculated using the mean decrease in impurity.

In *Step 6*, we plotted the cumulative importance of all the features. Using this plot, we could decide if we wanted to reduce the number of features in the model to account for a certain percentage of total importance. By doing so, we could potentially decrease the model's training time.

In *Step 7*, we calculated the permutation feature importance using the `permutation_importance` function available in `scikit-learn`. We decided to use recall as the scoring metric and set the `n_repeats` argument to 25, so the algorithm reshuffled each feature 25 times. The output of the procedure is a dictionary containing three elements: the raw feature importances, the average value per feature, and the corresponding standard deviation. Additionally, while using `permutation_importance` we can evaluate multiple metrics at once by providing a list of selected metrics.



We decided to use the `scikit-learn` implementation of permutation feature importance. However, there are alternative options available, for example, in the `rfpimp` or `eli5` libraries. The former also contains the drop-column feature importance.

In *Step 8*, we calculated and evaluated the permutation feature importance, this time using the test set.

We have mentioned in the introduction that permutation importance can return unreliable scores when our dataset has correlated features, that is, the importance score will be spread across the correlated features. We could try the following approaches to overcome this issue:

- Permute groups of correlated features together. `rfpimp` offers such functionality in the `importances` function.
- We could use hierarchical clustering on the features' Spearman's rank correlations, pick a threshold, and then only keep a single feature from each of the identified clusters.

In *Step 9*, we defined a function for calculating the drop-column feature importance. First, we trained and evaluated the baseline model using all features. As the scoring metric, we chose recall. Then, we used the `clone` function of `scikit-learn` to create a copy of the model with the exact same specification as the baseline one. We then iteratively trained the model on a dataset without one feature, calculated the selected evaluation metric, and stored the difference in scores.

In *Step 10*, we applied the drop-column feature importance function and plotted the results, both the most and least important features.

There's more...

We have mentioned that the default feature importance of scikit-learn's Random Forest is the MDI/Gini importance. It is also worth mentioning that the popular boosting algorithms (which we mentioned in the *Exploring ensemble classifiers* recipe) also adapted the `feature_importances_` attribute of the fitted model. However, they use different metrics of feature importance, depending on the algorithm.

For XGBoost, we have the following possibilities:

- `weight`—measures the number of times a feature is used to split the data across all trees. Similar to the Gini importance, however, it does not take into account the number of samples.
- `gain`—measures the average gain of the feature when it is used in trees. Intuitively we can think of it as the Gini importance measure, where Gini impurity is replaced by the objective of the gradient boosting model.
- `cover`—measures the average coverage of the feature when it is used in trees. Coverage is defined as the number of samples affected by the split.

The `cover` method can overcome one of the potential issues of the `weight` approach—simply counting the number of splits may be misleading, as some splits might affect just a few observations, and are therefore not really relevant.

For LightGBM, we have the following possibilities:

- `split`—measures the number of times the feature is used in a model
- `gain`—measures the total gains of splits that use the feature

See also

Additional resources are available here:

- Altmann, A., Tološi, L., Sander, O., & Lengauer, T. 2010. “Permutation importance: a corrected feature importance measure.” *Bioinformatics*, 26(10): 1340–1347.
- Louppe, G. 2014. “Understanding random forests: From theory to practice.” *arXiv preprint arXiv:1407.7502*.
- Molnar, C. 2020. *Interpretable Machine Learning*: <https://christophm.github.io/interpretable-ml-book/>
- Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. 2009. *The elements of statistical learning: data mining, inference, and prediction*, 2: 1–758. New York: Springer.
- Hooker, G., Mentch, L., & Zhou, S. 2021. “Unrestricted permutation forces extrapolation: variable importance requires at least one more model, or there is no free variable importance.” *Statistics and Computing*, 31(6): 1–16.
- Parr, T., Turgutlu, K., Csiszar, C., & Howard, J. 2018. Beware default random forest importances. March 26, 2018. <https://explained.ai/rf-importance/>.
- Strobl, C., Boulesteix, A. L., Kneib, T., Augustin, T., & Zeileis, A. 2008. “Conditional variable importance for random forests.” *BMC Bioinformatics*, 9(1): 307.
- Strobl, C., Boulesteix, A. L., Zeileis, A., & Hothorn, T. 2007. “Bias in random forest variable importance measures: Illustrations, sources and a solution.” *BMC bioinformatics*, 8(1): 1–21.

Exploring feature selection techniques

In the previous recipe, we saw how to evaluate the importance of features used for training ML models. We can use that knowledge to carry out feature selection, that is, keeping only the most relevant features and discarding the rest.

Feature selection is a crucial part of any machine learning project. First, it allows us to remove features that are either completely irrelevant or are not contributing much to a model's predictive capabilities. This can benefit us in multiple ways. Probably the most important benefit is that such unimportant features can actually negatively impact the performance of our model as they introduce noise and contribute to overfitting. As we have already established—*garbage in, garbage out*. Additionally, fewer features can often be translated into a shorter training time and help us avoid the curse of dimensionality.

Second, we should follow Occam's razor and keep our models simple and explainable. When we have a moderate number of features, it is easier to explain what is actually happening in the model. This can be crucial for the ML project's adoption by the stakeholders.

We have already established the *why* of feature selection. Now it is time to explore the *how*. On a high level, feature selection methods can be grouped into three categories:

- **Filter methods**—a generic set of univariate methods that specify a certain statistical measure and then filter the features based on it. This group does not incorporate any specific ML algorithm, hence it is characterized by (usually) lower computation time and is less prone to overfitting. A potential drawback of this group is that the methods evaluate the relationship between the target and each of the features individually. This can lead to them overlooking important relationships between the features. Examples include correlation, chi-squared test, analysis of variance (ANOVA), information gain, variance thresholding, and so on.
- **Wrapper methods**—this group of approaches considers feature selection a search problem, that is, it uses certain procedures to repeatedly evaluate a specific ML model with a different set of features to find the optimal set. It is characterized by the highest computational costs and the highest possibility of overfitting. Examples include forward selection, backward elimination, stepwise selection, recursive feature elimination, and so on.
- **Embedded methods**—this set of methods uses ML algorithms that have built-in feature selection, for example, Lasso with its regularization or Random Forest. By using these implicit feature selection methods, the algorithms try to prevent overfitting. In terms of computational complexity, this method is usually somewhere between the filter and wrapper groups.

In this recipe, we will apply a selection of feature selection methods to the credit card fraud dataset. We believe it provides a good example, especially given a lot of the features are anonymized and we do not know the exact meaning behind them. Hence, it is also likely that some of them do not really contribute much to the model's performance.

Getting ready

In this recipe, we will be using the credit card fraud dataset that we introduced in the *Investigating different approaches to handling imbalanced data* recipe. For convenience, we have included all the necessary preparation steps in this section from the accompanying Jupyter notebook.



Another interesting challenge to applying feature selection methods would be BNP Paribas Cardif Claims Management (the dataset is available at Kaggle—a link is provided in the *See also* section). Similar to the dataset used in this recipe, it contains 131 anonymized features.

How to do it...

Execute the following steps to experiment with various feature selection methods:

1. Import the libraries:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import recall_score
from sklearn.feature_selection import (RFE, RFECV,
                                         SelectKBest,
                                         SelectFromModel,
                                         mutual_info_classif)
from sklearn.model_selection import StratifiedKFold
```

2. Train the benchmark model:

```
rf = RandomForestClassifier(random_state=RANDOM_STATE,
                           n_jobs=-1)
rf.fit(X_train, y_train)

recall_train = recall_score(y_train, rf.predict(X_train))
recall_test = recall_score(y_test, rf.predict(X_test))
print(f"Recall score training: {recall_train:.4f}")
print(f"Recall score test: {recall_test:.4f}")
```

Executing the snippet generates the following output:

```
Recall score training: 1.0000
Recall score test: 0.8265
```

Looking at the recall scores, the model is clearly overfitted to the training data. Normally, we should try to address this. However, to keep the exercise simple we assume that the model is good enough to proceed.

3. Select the best features using Mutual Information:

```
scores = []
n_features_list = list(range(2, len(X_train.columns)+1))

for n_feat in n_features_list:
    print(f"Keeping {n_feat} most important features")
    mi_selector = SelectKBest(mutual_info_classif, k=n_feat)
    X_train_new = mi_selector.fit_transform(X_train, y_train)
    X_test_new = mi_selector.transform(X_test)

    rf.fit(X_train_new, y_train)
    recall_scores = [
        recall_score(y_train, rf.predict(X_train_new)),
        recall_score(y_test, rf.predict(X_test_new))
    ]
    scores.append(recall_scores)

mi_scores_df = pd.DataFrame(
    scores,
    columns=["train_score", "test_score"],
    index=n_features_list
)
```

Using the next snippet, we plot the results:

```
(  
    mi_scores_df["test_score"]  
    .plot(kind="bar",  
          title="Feature selection using Mutual Information",  
          xlabel="# of features",  
          ylabel="Recall (test set)")  
)
```

Executing the snippet generates the following plot:

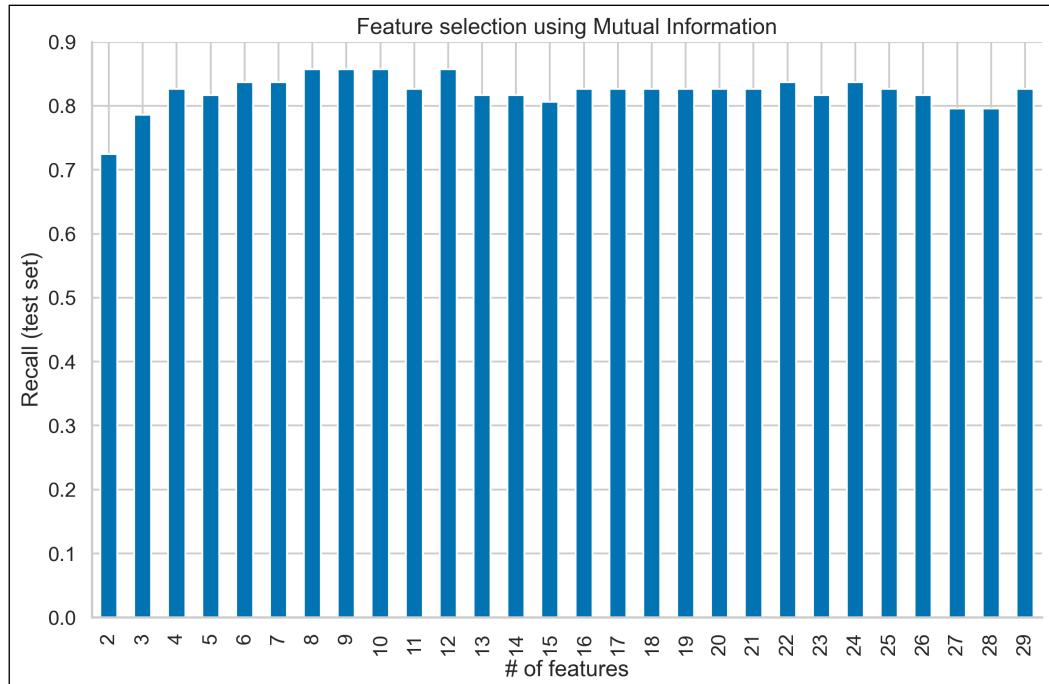


Figure 14.32: Performance of the model depending on the number of selected features.
Features are selected using the Mutual Information criterion

By inspecting the figure, we can see that we achieved the best recall score on the test set using 8, 9, 10, and 12 features. As simplicity is desired, we decided to choose 8. Using the following snippet, we extract the names of the 8 most important features:

```
mi_selector = SelectKBest(mutual_info_classif, k=8)
mi_selector.fit(X_train, y_train)
print(f"Most importance features according to MI: {mi_selector.get_
feature_names_out()}")
```

Executing the snippet returns the following output:

```
Most importance features according to MI: ['V3' 'V4' 'V10' 'V11' 'V12'
'V14' 'V16' 'V17']
```

4. Select the best features using MDI feature importance, retrain the model, and evaluate its performance:

```
rf_selector = SelectFromModel(rf)
rf_selector.fit(X_train, y_train)

mdi_features = X_train.columns[rf_selector.get_support()]
rf.fit(X_train[mdi_features], y_train)
recall_train = recall_score(
    y_train, rf.predict(X_train[mdi_features]))
)
recall_test = recall_score(y_test, rf.predict(X_test[mdi_features]))
print(f"Recall score training: {recall_train:.4f}")
print(f"Recall score test: {recall_test:.4f}")
```

Executing the snippet generates the following output:

```
Recall score training: 1.0000
Recall score test: 0.8367
```

Using the following snippet, we extract the threshold used for feature selection and the most relevant features:

```
print(f"MDI importance threshold: {rf_selector.threshold_:.4f}")
print(f"Most importance features according to MI: {rf_selector.get_
feature_names_out()}")
```

This generates the following output:

```
MDI importance threshold: 0.0345
Most importance features according to MDI: ['V10' 'V11' 'V12' 'V14' 'V16'
'V17']
```

The threshold value corresponds to the average feature importance of the RF model.

Using a loop similar to the one in *Step 3*, we can generate a bar chart showing the model's performance depending on the number of features kept in the model. We iteratively select the top k features based on the MDI. To avoid repetition, we do not include the code here (it is available in the accompanying Jupyter notebook). By analyzing the figure, we can see that the model achieved the best score with 10 features, which is more than in the previous approach.

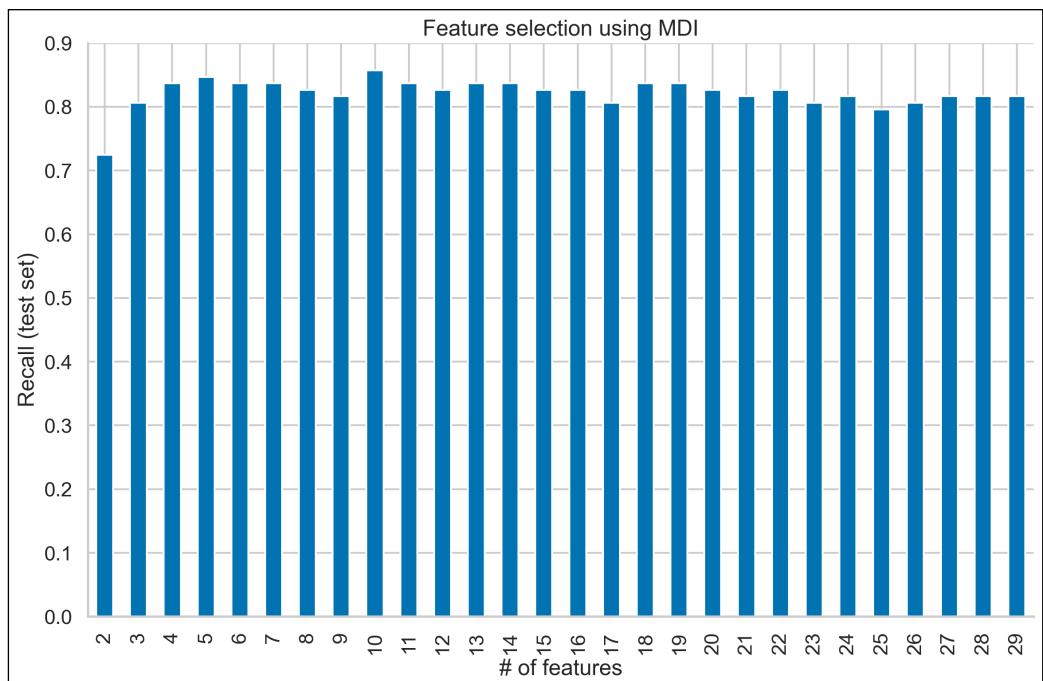


Figure 14.33: Performance of the model depending on the number of selected features.
Features are selected using the Mean Decrease in Impurity feature importance

- Select the best 10 features using Recursive Feature Elimination:

```
rfe = RFE(estimator=rf, n_features_to_select=10, verbose=1)
rfe.fit(X_train, y_train)
```

In order to avoid repetition, we present the most important features and the accompanying scores without the code, as it is almost identical to what we have covered in the previous steps:

```
Most importance features according to RFE: ['V4' 'V7' 'V9' 'V10' 'V11'
'V12' 'V14' 'V16' 'V17' 'V26']
Recall score training: 1.0000
Recall score test: 0.8367
```

6. Select the best features using Recursive Feature Elimination with cross-validation:

```
k_fold = StratifiedKFold(5, shuffle=True, random_state=42)

rfe_cv = RFECV(estimator=rf, step=1,
                 cv=k_fold,
                 min_features_to_select=5,
                 scoring="recall",
                 verbose=1, n_jobs=-1)
rfe_cv.fit(X_train, y_train)
```

Below we present the outcome of the feature selection:

```
Most importance features according to RFECV: ['V1' 'V4' 'V6' 'V7' 'V9'
'V10' 'V11' 'V12' 'V14' 'V15' 'V16' 'V17' 'V18'
'V20' 'V21' 'V26']
Recall score training: 1.0000
Recall score test: 0.8265
```

This approach resulted in the selection of 16 features. Overall, 6 features appeared in each of the considered approaches: V10, V11, V12, V14, V16, and V17.

Additionally, using the following snippet we can visualize the cross-validation scores, that is, what the average recall of the 5 folds was for each of the considered numbers of retained features. We had to add 5 to the index of the DataFrame, as we chose to retain a minimum of 5 features in the RFECV procedure:

```
cv_results_df = pd.DataFrame(rfe_cv.cv_results_)
cv_results_df.index += 5

(
    cv_results_df["mean_test_score"]
    .plot(title="Average CV score over iterations",
          xlabel="# of features retained",
          ylabel="Avg. recall")
)
```

Executing the snippet generates the following plot:

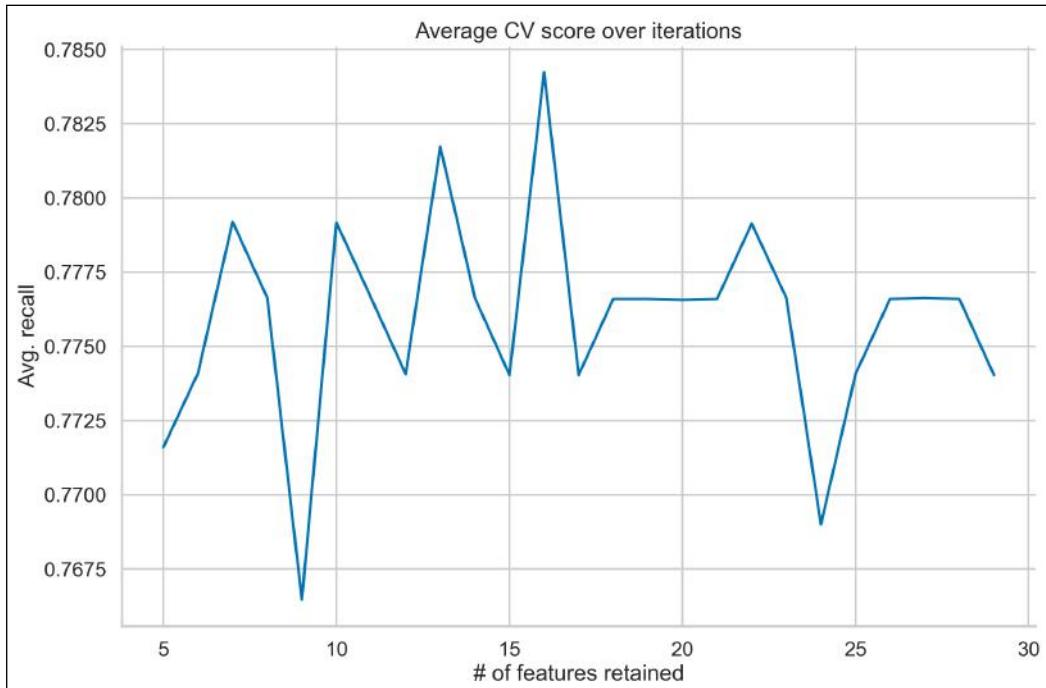


Figure 14.34: Average CV score for each step of the RFE procedure

Inspecting the figure confirms that the highest average recall was obtained using 16 features.

While evaluating the benefits of feature selection, we should consider two scenarios. In the more obvious one, the performance of the model improves when we remove some of the features. This does not need any further explanation. The second scenario is more interesting. After removing features, we can end up with a very similar performance to the initial one or slightly worse. However, this does not necessarily mean that we have failed. Consider a case in which we removed ~60% of the features while keeping the same performance. This could already be a major improvement that—depending on the dataset and model—can potentially reduce the training time by hours or days. Additionally, such a model would be easier to interpret.

How it works...

After importing the required libraries, we trained a benchmark Random Forest classifier and printed the recall score from the training and test sets.

In *Step 3*, we applied the first of the considered feature selection approaches. It was an example of the univariate *filter* category of feature selection techniques. As the statistical criterion, we used the Mutual Information score. To calculate the metric, we used the `mutual_info_classif` function from `scikit-learn`, which is capable of working with a categorical target and numerical features only. Hence, any categorical features need to be appropriately encoded beforehand. Fortunately, we only have continuous numerical features in this dataset.



The **Mutual Information (MI)** score of two random variables is a measure of the mutual dependence between those variables. When the score is equal to zero, the two variables are independent. The higher the score, the higher the dependency between the variables. In general, calculating the MI requires knowledge of the probability distributions of each of the features, which we do not usually know. That is why the `scikit-learn` implementation uses a nonparametric approximation based on k-Nearest Neighbors distances. One of the advantages of using MI is that it can capture nonlinear relationships between the features.

Next, we combined the MI criterion with the `SelectKBest` class, which allows us to select the k best features determined by an arbitrary metric. Using this approach, we almost never know upfront how many features we would like to keep. Hence, we iterated over all the possible values (from 2 to 29, where the latter is the total number of features in the dataset). The `SelectKBest` class employs the familiar `fit/transform` approach. Within each iteration, we fitted the class to the training data (both features and the target are required for this step) and then transformed the training and test sets. The transformation resulted in keeping only the k most important features according to the MI criterion. Then, we once again fitted the Random Forest classifier using only the selected features and recorded the relevant recall scores.

`scikit-learn` allows us to easily use different metrics together with the `SelectKBest` class. For example, we could use the following scoring functions:

- `f_classif`—the ANOVA F-value estimating the degree of linear dependency between two variables. The F statistic is calculated as the ratio of between-group variability to the within-group variability. In this case, the group is simply the class of the target. A potential drawback of this method is that it only accounts for linear relationships.
- `chi2`—the chi-squared statistics. This metric is only suitable for non-negative features such as Booleans or frequencies, or more generally, for categorical features. Intuitively, it evaluates if a feature is independent of the target. If that is the case, it is also uninformative when it comes to classifying the observations.

Aside from selecting the k best features, the `feature_selection` module of `scikit-learn` also offers classes that allow choosing features based on the percentile of the highest scores, a false positive rate test, an estimated false discovery rate, or a family-wise error rate.

In *Step 4*, we explored an example of the *embedded* feature selection techniques. In this group, feature selection is performed as part of the model building phase. We used the `SelectFromModel` class to select the best features based on the model's built-in feature importance metric (in this case, the MDI feature importance). When instantiating the class, we can provide the `threshold` argument to determine the threshold used to select the most relevant features. Features with weights/coefficients above that threshold would be kept in the model. We can also use the "mean" (default one) and "median" keywords to use the mean/median values of all feature importances as the threshold. We can also combine those keywords with scaling factors, for example, "1.5*mean". Using the `max_features` argument, we can determine the maximum number of features we allow to be selected.



The `SelectFromModel` class works with any estimator that has either the `feature_importances_` (for example, Random Forest, XGBoost, LightGBM, and so on) or `coef_` (for example, Linear Regression, Logistic Regression, and Lasso) attribute.

In this step, we demonstrated two approaches to recovering the selected features. The first one is the `get_support` method, which returns a list with Boolean flags indicating whether the given feature was selected. The second one is the `get_feature_names_out` method, which directly returns the names of the selected features. While fitting the Random Forest classifier, we manually selected the columns of the training dataset. However, we could have also used the `transform` method of the fitted `SelectFromModel` class to automatically extract only the relevant features as a numpy array.

In *Step 5*, we used an example of the *wrapper* methods. **Recursive Feature Elimination (RFE)** is an algorithm that recursively trains an ML model, calculates the feature importances (via `coef_` or `feature_importances_`), and drops the least important feature or features.

The process starts by training the model using all the features. Then, the least important feature or features are pruned from the dataset. Next, the model is trained again with the reduced feature set, and the least important features are again eliminated. The process is repeated until it reaches the desired number of features. While instantiating the `RFE` class, we provided the Random Forest estimator together with the number of features to select. Additionally, we could provide the `step` argument, which determined how many features to eliminate during each iteration.



RFE can be a computationally expensive algorithm to run, especially with a large feature set and cross-validation. Hence, it might be a good idea to apply some other feature selection technique before using RFE. For example, we could use the filtering approach and remove some of the correlated features.

As we have mentioned before, we rarely know the optimal number of features upfront. That is why in *Step 6* we try to account for that drawback. By combining RFE with cross-validation, we can automatically determine the optimal number of features to keep using the RFE procedure. To do so, we used the `RFECV` class and provided some additional inputs. We had to specify the cross-validation scheme (5-fold stratified CV, as we are dealing with an imbalanced dataset), the scoring metric (recall), and the minimum number of features to retain. For the last argument, we arbitrarily chose 5.

Lastly, to explore the CV scores in more depth, we accessed the cross-validation scores for each fold using the `cv_results_` attribute of the fitted RFECV class.

There's more...

Some of the other available approaches

We have already mentioned quite a few univariate filter methods. Some other notable ones include:

- **Variance thresholding**—this method simply removes features with variance lower than a specified threshold. Thus, it can be used to remove constant and quasi-constant features. The latter ones are those that have very little variability as almost all the values are identical. By definition, this method does not look at the target value, only at the features.
- **Correlation-based**—there are multiple ways to measure correlation, hence we will only focus on the general logic of this approach. First, we determine the correlation between the features and the target. We can choose a threshold above which we want to keep the features for modeling.

Then, we should also consider removing features that are highly correlated among themselves. We should identify such groups and then leave only one feature from each of the groups in our dataset. Alternatively, we could use the **Variance Inflation Factor (VIF)** to determine multicollinearity and drop features based on high VIF values. VIF is available in `statsmodels`.



We did not consider using correlation as a criterion in this recipe, as the features in the credit card fraud dataset are the outcomes of PCA. Hence, by definition they are orthogonal, that is, uncorrelated.

There are also multivariate filter methods available. For example, **Maximum Relevance Minimum Redundancy (MRMR)** is a family of algorithms that attempts to identify a subset of features that have high relevance with respect to the target variable, while having a small redundancy with each other.

We could also explore the following *wrapper* techniques:

- **Forward feature selection**—we start with no features. We test each of the features separately and see which one most improves the model. We add that feature to our feature set. Then, we sequentially train models with a second feature added. Similarly, at this step, we again test all the remaining features individually. We select the best one and add it to the selected pool. We continue adding features one at a time until we reach a stopping criterion (max number of features or no further improvement). Traditionally, the feature to be added was based on the features' p-values. However, modern libraries use the improvement on a cross-validated metric of choice as the selection criterion.
- **Backward feature selection**—similar to the previous approach, but we start with all the features in our set and sequentially remove one feature at a time until there is no further improvement (or all features are statistically significant). This method differs from RFE as it does not use the coefficients or feature importances to select the features to be removed. Instead, it optimizes for the performance improvement measured by the difference in the cross-validated score.

- **Exhaustive feature selection**—simply speaking, in this brute-force approach we try all the possible combinations of the features. Naturally, this is the most computationally expensive of the wrapper techniques, as the number of feature combinations to be tested grows exponentially with the number of features. For example, if we had 3 features, we would have to test 7 combinations. Assume we have features a, b, and c. We would have to test the following combinations: [a, b, c, ab, ac, bc, abc].
- **Stepwise selection**—a hybrid approach combining forward and backward feature selection. The process starts with zero features and adds them one by one using the lowest significant p-value. At each addition step, the procedure also checks if any of the current features are statistically insignificant. If that is the case, they are dropped from the feature set and the algorithm continues to the next addition step. The procedure allows the final model to have only statistically significant features.

The first two approaches are implemented in `scikit-learn`. Alternatively, you can find all four of them in the `m1xtend` library.

We should also mention a few things to keep in mind about the *wrapper* techniques presented above:

- The optimal number of features depends on the ML algorithm.
- Due to their iterative nature, they are able to detect certain interactions between the features.
- These methods usually provide the best performing subset of features for a given ML algorithm.
- They come at the highest computational cost, as they operate greedily and retrain the model multiple times.

As the last wrapper method, we will mention the **Boruta** algorithm. Without going into too much detail, it creates a set of shadow features (permuted duplicates of the original features) and selects features using a simple heuristic: a feature is useful if it is doing better than the best of the randomized features. The entire process is repeated multiple times before the algorithm returns the best set of features. The algorithm is compatible with ML models from the `ensemble` module of `scikit-learn` and algorithms such as XGBoost and LightGBM. For more details on the algorithm, please refer to the paper mentioned in the *See also* section. The Boruta algorithm is implemented in the `boruta` library.

Lastly, it is worth mentioning that we can also combine multiple feature selection approaches to improve their reliability. For example, we could select features using a few approaches and then ultimately select the ones that appeared in all or most of them.

Combining feature selection and hyperparameter tuning

As we have already established, we do not know the optimal number of features to keep in advance. Hence, we might want to combine feature selection with hyperparameter tuning and treat the number of features to keep as another hyperparameter.

We can easily do so using pipelines and GridSearchCV from scikit-learn:

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

pipeline = Pipeline(
    [
        ("selector", SelectKBest(mutual_info_classif)),
        ("model", rf)
    ]
)

param_grid = {
    "selector__k": [5, 10, 20, 29],
    "model__n_estimators": [10, 50, 100, 200]
}

gs = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    n_jobs=-1,
    scoring="recall",
    cv=k_fold,
    verbose=1
)

gs.fit(X_train, y_train)
print(f"Best hyperparameters: {gs.best_params_}")
```

Executing the snippet returns the best set of hyperparameters:

```
Best hyperparameters: {'model__n_estimators': 50, 'selector__k': 20}
```



When combining filter feature selection methods with cross-validation, we should do the filtering within the cross-validation procedure. Otherwise, we are selecting the features using all the available observations and introducing bias.

One thing to keep in mind is that the features selected within various folds of the cross-validation can be different. Let's consider an example of a 5-fold cross-validation procedure that keeps 3 features. It can happen that in some of the 5 cross-validation rounds, the 3 selected features might not overlap. However, they should not be too different, as we assume that the overall patterns in the data and the distribution of the features are very similar across folds.

See also

Additional references on the topic:

- Bommert, A., Sun, X., Bischl, B., Rahnenführer, J., & Lang, M. 2020. “Benchmark for filter methods for feature selection in high-dimensional classification data.” *Computational Statistics & Data Analysis*, 143: 106839.
- Ding, C., & Peng, H. 2005. “Minimum redundancy feature selection from microarray gene expression data.” *Journal of bioinformatics and computational biology*, 3(2): 185–205.
- Kira, K., & Rendell, L. A. 1992. A practical approach to feature selection. In *Machine learning proceedings*, 1992: 249–256. Morgan Kaufmann.
- Kira, K., & Rendell, L. A. 1992, July. The feature selection problem: Traditional methods and a new algorithm. In *Aaai*, 2(1992a): 129-134.
- Kuhn, M., & Johnson, K. 2019. *Feature engineering and selection: A practical approach for predictive models*. CRC Press.
- Kursa M., & Rudnicki W. Sep. 2010. “Feature Selection with the Boruta Package” *Journal of Statistical Software*, 36(11): 1-13.
- Urbanowicz, RJ., et al. 2018. “Relief-based feature selection: Introduction and review.” *Journal of biomedical informatics*, 85: 189–203.
- Yu, L., & Liu, H. 2003. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings of the 20th international conference on machine learning (ICML-03)*: 856–863.
- Zhao, Z., Anand, R., & Wang, M. 2019, October. Maximum relevance and minimum redundancy feature selection methods for a marketing machine learning platform. In *2019 IEEE international conference on data science and advanced analytics (DSAA)*: 442–452. IEEE.

You can find the additional dataset mentioned in the *Getting ready* section here:

- <https://www.kaggle.com/competitions/bnp-paribas-cardif-claims-management>

Exploring explainable AI techniques

In one of the previous recipes, we looked into feature importance as one of the means of getting a better understanding of how the models work under the hood. While this might be quite a simple task in the case of linear regression, it gets increasingly difficult with the complexity of the models.

One of the big trends in the ML/DL field is **explainable AI (XAI)**. It refers to various techniques that allow us to better understand the predictions of black box models. While the current XAI approaches will not turn a black box model into a fully interpretable one (or a white box), they will definitely help us better understand why the model returns certain predictions for a given set of features.

Some of the benefits of having explainable AI models are as follows:

- Builds trust in the model—if the model’s reasoning (via its explanation) matches common sense or the beliefs of human experts, it can strengthen the trust in the model’s predictions
- Facilitates the model’s or project’s adoption by business stakeholders

- Gives insights useful for human decision-making by providing reasoning for the model's decision process
- Makes debugging easier
- Can steer the direction of future data gathering or feature engineering

Before mentioning the particular XAI techniques, it is worth clarifying the difference between interpretability and explainability. **Interpretability** can be considered a stronger version of explainability. It offers a causality-based explanation of a model's predictions. On the other hand, **explainability** is used to make sense of the predictions made by black box models, which cannot be interpretable. In particular, XAI techniques can be used to explain what is going on in the model's prediction process, but they are unable to causally prove why a certain prediction has been made.

In this recipe, we cover three XAI techniques. See the *There's more...* section for a reference to more of the available approaches.

The first technique is called **Individual Conditional Expectation (ICE)** and it is a local and model-agnostic approach to explainability. The *local* part refers to the fact that this technique describes the impact of feature(s) at the observation level. ICE is most frequently presented in a plot and depicts how an observation's prediction changes as a result of a change in a given feature's value.

To obtain the ICE values for a single observation in our dataset and one of its features, we have to create multiple copies of that observation. In all of them, we keep the values of other features (except the considered one) constant, while replacing the value of the feature of interest with the values from a grid. Most commonly, the grid consists of all the distinct values of that feature in the entire dataset (for all observations). Then, we use the (black box) model to make predictions for each of the modified copies of the original observation. Those predictions are plotted as the ICE curve.

Advantages:

- It is easy to calculate and intuitive to understand what the curves represent.
- ICE can uncover heterogeneous relationships, that is, when a feature has a different direction of impact on the target, depending on the intervals of the explored feature's values.

Disadvantages:

- We can meaningfully display only one feature at a time.
- Plotting many ICE curves (for multiple observations) can make the plot overcrowded and hard to interpret.
- ICE assumes independence of features—when features are correlated, some points in the curve might actually be invalid data points (either very unlikely or simply impossible) according to the joint feature distribution.

The second approach is called the **Partial Dependence Plot (PDP)** and is heavily connected to ICE. It is also a model-agnostic method; however, it is a global one. It means that PDP describes the impact of feature(s) on the target in the context of the entire dataset.

PDP presents the marginal effect of a feature on the prediction. Intuitively, we can think of partial dependence as a mapping of the expected response of the target as a function of the feature of interest. It can also show whether the relationship between the feature and the target is linear or nonlinear. In terms of calculating the PDP, it is simply the average of all the ICE curves.

Advantages:

- Similar to ICE, it is easy to calculate and intuitive to understand what the curves represent.
- If the feature of interest is not correlated with other features, the PDP then perfectly represents how the selected feature impacts the prediction (on average).
- The calculation for the PDPs has a causal interpretation (within the model)—by observing the changes in prediction caused by the changes to one of the features, we analyze the causal relationship between the two.

Disadvantages:

- PDPs also assume the independence of features.
- PDPs can obscure heterogeneous relationships created by interactions. For example, we could observe a linear relationship between the target and a certain feature. However, the ICE curves might show that there are exceptions to that pattern, for example, where the target remains constant in some ranges of the feature.
- PDPs can be used to analyze, at most, two features at a time.

The last of the XAI techniques we cover in this recipe is called **SHapley Additive exPlanations (SHAP)**. It is a model-agnostic framework for explaining predictions using a combination of game theory and local explanations.

The exact methodology and calculations involved in this method are outside of the scope of this book. We can briefly mention that **Shapley values** are a method used in game theory that involves a fair distribution of both gains and costs to players cooperating in a game. As each player contributes differently to the coalition, the Shapley value makes sure that each participant gets a fair share, depending on how much they contributed.

We could compare it to the ML setting, in which features are the players, the cooperative game is creating the ML model's prediction, and the payoff is the difference between the average prediction of the instance minus the average prediction of all instances. Hence, the interpretation of a Shapley value for a certain feature is as follows: the value of the feature contributed x to the prediction of this observation, compared to the average prediction for the dataset.

Having covered the Shapley values, it is time to explain what SHAP is. It is an approach to explaining the outputs of any ML/DL model. SHAP combines optimal credit allocation with local explanations, using Shapley values (originating from game theory) and their extensions.

SHAP offers the following:

- It is a computationally efficient and theoretically robust method of calculating Shapley values for ML models (ideally having trained the model only once).

- KernelSHAP—an alternative, kernel-based estimation method for estimating Shapley values. It was inspired by local surrogate models.
- TreeSHAP—an efficient estimation method for tree-based models.
- Various global interpretation methods based on aggregations of Shapley values.



To get a better understanding of SHAP, it is recommended to also get familiar with LIME. Please refer to the *There's more...* section for a brief description.

Advantages:

- Shapley values have a solid theoretical background (axioms of efficiency, symmetry, dummy, and additivity). Lundberg *et al.* (2017) explain minor discrepancies between those axioms in the context of Shapley values and their counterpart properties of the SHAP values, that is, local accuracy, missingness, and consistency.
- Thanks to the efficiency property, SHAP might be the only framework in which the prediction is fairly distributed among the feature values.
- SHAP offers global interpretability—it shows feature importance, feature dependence, interactions, and an indication of whether a certain feature has a positive or negative impact on the model's predictions.
- SHAP offers local interpretability—while many techniques only focus on aggregate explainability, we can calculate SHAP values for each individual prediction to learn how features contribute to that particular prediction.
- SHAP can be used to explain a large variety of models, including linear models, tree-based models, and neural networks.
- TreeSHAP (the fast implementation for tree-based models) makes it feasible to use the approach for real-life use cases.

Disadvantages:

- Computation time—the number of possible combinations of the features increases exponentially with the number of considered features, which in turn increases the time of calculating SHAP values. That is why we have to revert to approximations.
- Similar to permutation feature importance, SHAP values are sensitive to high correlations among features. If that is the case, the impact of such features on the model score can be split among those features in an arbitrary way, leading us to believe that they are less important than if their impacts remained undivided. Also, correlated features might result in using unrealistic/impossible combinations of features.
- As Shapley values do not offer a prediction model (such as in the case of LIME), they cannot be used to make statements about how a change in the inputs corresponds to a change in the prediction. For example, we cannot state that “if the value of feature Y was higher by 50 units, then the predicted probability would increase by 1 percentage point.”

- KernelSHAP is slow and, similarly to other permutation-based interpretation methods, ignores dependencies between features.

Getting ready

In this recipe, we will be using the credit card fraud dataset that we introduced in the *Investigating different approaches to handling imbalanced data* recipe. For convenience, we have included all the necessary preparation steps in this section of the accompanying Jupyter notebook.

How to do it...

Execute the following steps to investigate various approaches to explaining the predictions of an XGBoost model trained on the credit card fraud dataset:

- Import the libraries:

```
from xgboost import XGBClassifier
from sklearn.metrics import recall_score
from sklearn.inspection import (partial_dependence,
                                 PartialDependenceDisplay)
import shap
```

- Train the ML model:

```
xgb = XGBClassifier(random_state=RANDOM_STATE,
                     n_jobs=-1)
xgb.fit(X_train, y_train)

recall_train = recall_score(y_train, xgb.predict(X_train))
recall_test = recall_score(y_test, xgb.predict(X_test))
print(f'Recall score training: {recall_train:.4f}')
print(f'Recall score test: {recall_test:.4f}')
```

Executing the snippet generates the following output:

```
Recall score training: 1.0000
Recall score test: 0.8163
```

We can conclude that the model is overfitted to the training data and ideally we should try to fix that by, for example, using stronger regularization while training the XGBoost model. To keep the exercise concise, we assume that the model is good to go for further analysis.



Similarly to investigating feature importance, we should first make sure that the model has satisfactory performance on the validation/test set before we start explaining its predictions.

3. Plot the ICE curves:

```
PartialDependenceDisplay.from_estimator(
    xgb, X_train, features=["V4"],
    kind="individual",
    subsample=5000,
    line_kw={"linewidth": 2},
    random_state=RANDOM_STATE
)
plt.title("ICE curves of V4")
```

Executing the snippet generates the following plot:

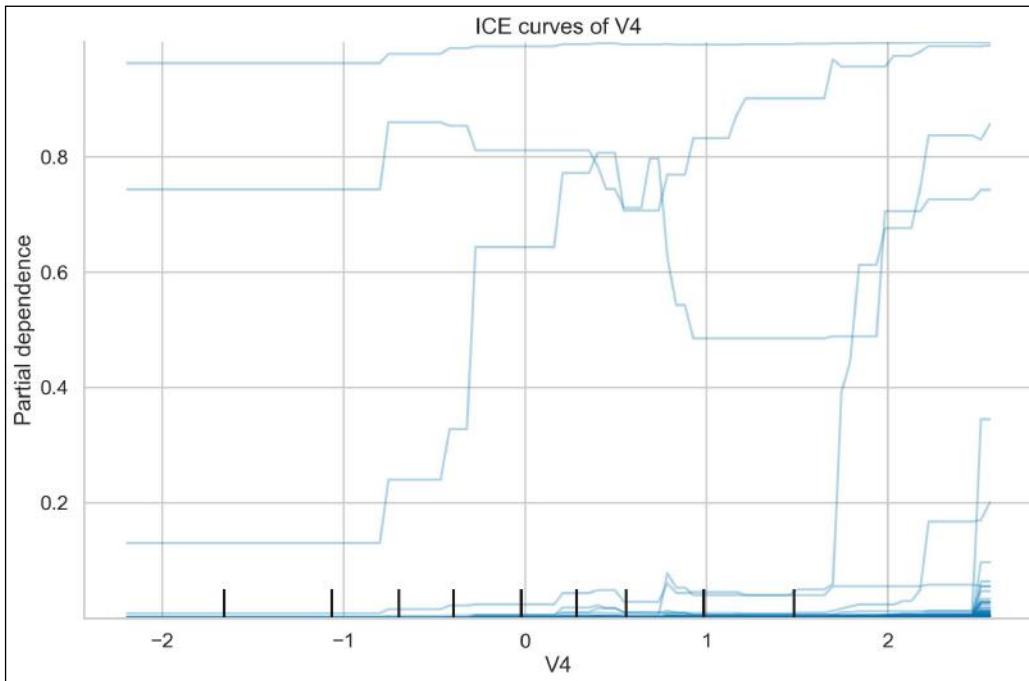


Figure 14.35: The ICE plot of the V4 feature, created using 5,000 random samples from the training data

Figure 14.35 presents the ICE curves for the V4 feature, calculated using 5,000 random observations from the training data. In the plot, we can see that the vast majority of the observations are located around 0, while a few of the curves show quite a significant change in predicted probability.

The black marks at the bottom of the plot indicate the percentiles of the feature values. By default, the ICE plot and PDP are constrained to the 5th and 95th percentiles of the feature values; however, we can change this using the `percentiles` argument.

A potential issue with the ICE curves is that it might be hard to see if the curves differ between observations, as they start at different predictions. A solution would be to center the curves at a certain point and display only the difference in the prediction compared to that point.

4. Plot the centered ICE curves:

```
PartialDependenceDisplay.from_estimator(  
    xgb, X_train, features=["V4"],  
    kind="individual",  
    subsample=5000,  
    centered=True,  
    line_kw={"linewidth": 2},  
    random_state=RANDOM_STATE  
)  
plt.title("Centered ICE curves of V4")
```

Executing the snippet generates the following plot:

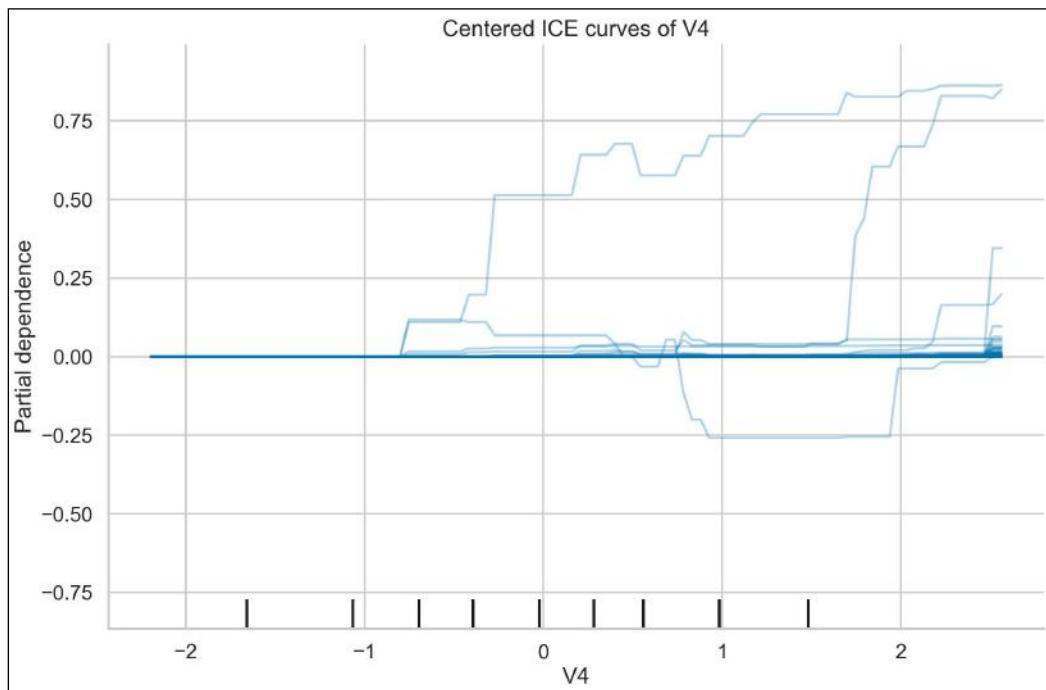


Figure 14.36: The centered ICE plot of the V4 feature, created using 5,000 random samples from the training data

The interpretation of the centered ICE curves is only slightly different. Instead of looking at the impact of changing the value of a feature on the prediction, we look at the relative change in the prediction, as compared to the average prediction. This way, it is easier to analyze the direction of the change in the predicted value.

5. Generate the Partial Dependence Plot:

```
PartialDependenceDisplay.from_estimator(  
    xgb, X_train,  
    features=[ "V4" ],  
    random_state=RANDOM_STATE  
)  
plt.title("Partial Dependence Plot of V4")
```

Executing the snippet generates the following plot:

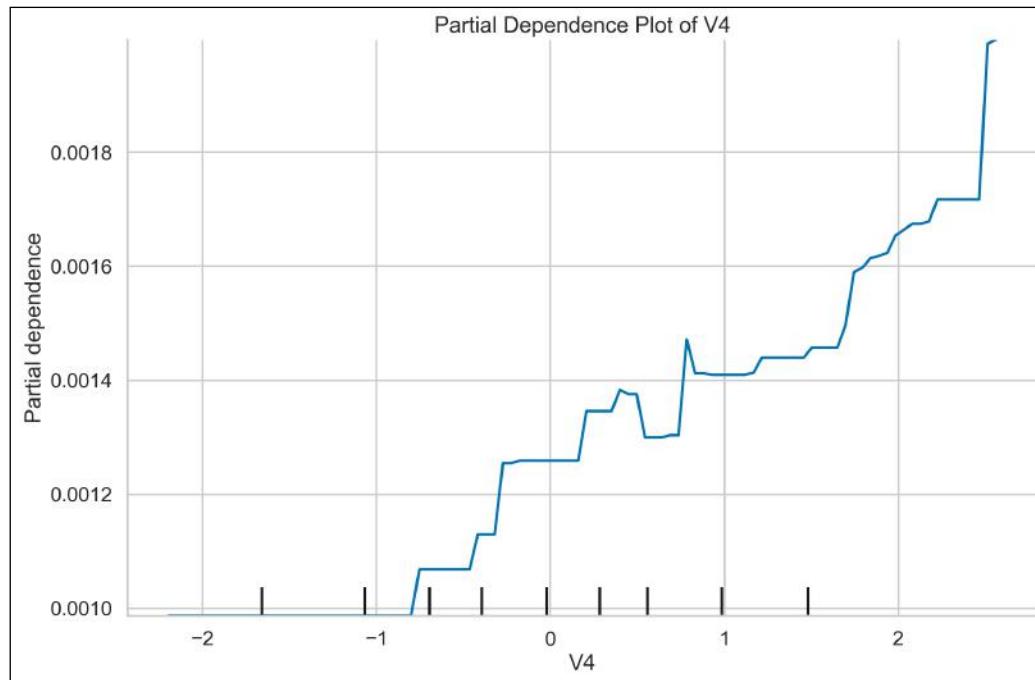


Figure 14.37: The Partial Dependence Plot of the V4 feature, prepared using the training data

By analyzing the plot, on average there seems to be a very small increase in the predicted probability with the increase of the V4 feature.



Similar to the ICE curves, we can also center the PDP.

To get some further insights, we can generate the PDP together with the ICE curves. We can do so using the following snippet:

```
PartialDependenceDisplay.from_estimator(  
    xgb, X_train, features=["V4"],  
    kind="both",  
    subsample=5000,  
    ice_lines_kw={"linewidth": 2},  
    pd_line_kw={"color": "red"},  
    random_state=RANDOM_STATE  
)  
plt.title("Partial Dependence Plot of V4, together with ICE curves")
```

Executing the snippet generates the following plot:

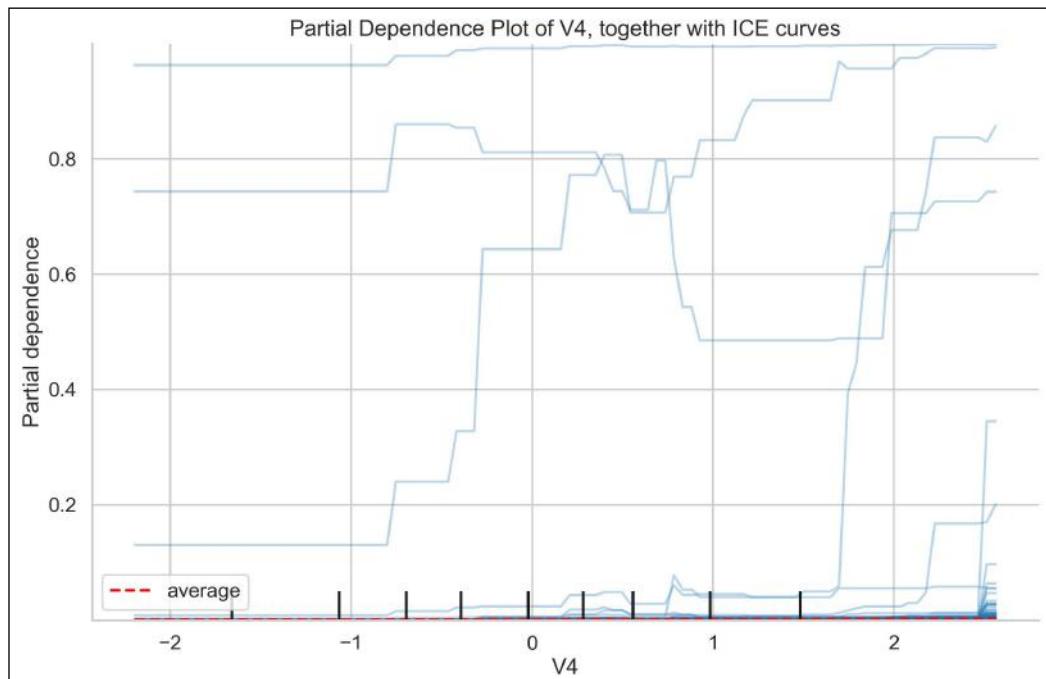


Figure 14.38: The Partial Dependence Plot of the V4 feature (prepared using the training data), together with the ICE curves

As we can see, the partial dependence (PD) line is almost horizontal at 0. Because of the differences in scale (please refer to *Figure 14.37*), the PD line is virtually meaningless in such a plot. To make the plot more readable or easier to interpret, we could try restricting the range of the y-axis using the `plt.ylim` function. This way, we would focus on the area with the majority of the ICE curves, while neglecting the few ones that are far away from the bulk of the curves. However, we should keep in mind that those outlier curves are also important for the analysis.

6. Generate the individual PDPs of two features and a joint one:

```
fig, ax = plt.subplots(figsize=(20, 8))

PartialDependenceDisplay.from_estimator(
    xgb,
    X_train.sample(20000, random_state=RANDOM_STATE),
    features=["V4", "V8", ("V4", "V8")],
    centered=True,
    ax=ax
)
ax.set_title("Centered Partial Dependence Plots of V4 and V8")
```

Executing the snippet generates the following plot:

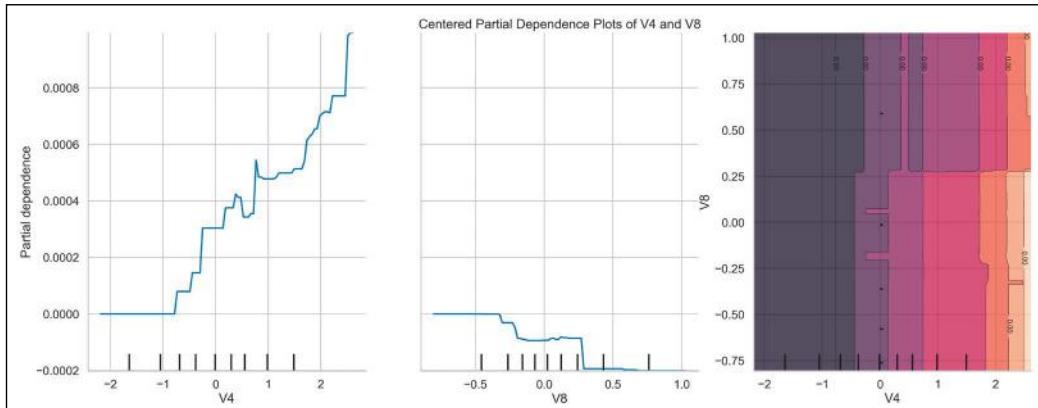


Figure 14.39: The centered Partial Dependence Plot of the V4 and V8 features, individually and jointly

By jointly plotting the PDPs of two features, we are able to visualize the interactions among them. By looking at *Figure 14.39* we could draw a conclusion that the V4 feature is more important, as most of the lines visible in the rightmost plot are perpendicular to the V4 axis and parallel to the V8 axis. However, there is some shift in the decision lines determined by the V8 feature, for example, around the 0.25 value.

7. Instantiate an explainer and calculate the SHAP values:

```
explainer = shap.TreeExplainer(xgb)
shap_values = explainer.shap_values(X)
explainer_X = explainer(X)
```

The `shap_values` object is a 284807 by 29 numpy array containing the calculated SHAP values.

8. Generate the SHAP summary plot:

```
shap.summary_plot(shap_values, X)
```

Executing the snippet generates the following plot:

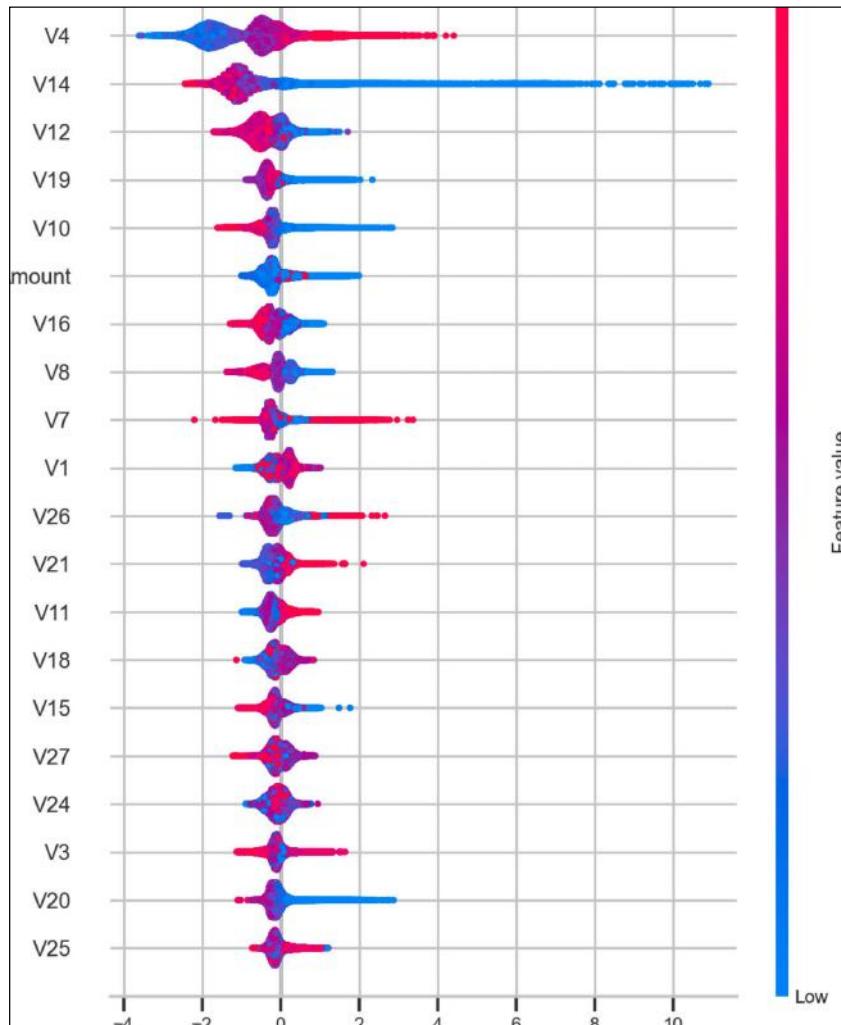


Figure 14.40: The summary plot calculated using SHAP values

When looking at the summary plot, we should be aware of the following:

- Features are sorted by the sum of the SHAP value magnitudes (absolute values) across all observations.
- The color of the points shows if that feature had a high or low value for that observation.
- The horizontal location on the plot shows whether the effect of that feature's value resulted in a higher or lower prediction.
- By default, the plots display the 20 most important features. We can adjust that using the `max_display` argument.
- Overlapping points are jittered in the y axis direction. Hence, we can get a sense of the distribution of the SHAP values per feature.
- An advantage of this type of plot over other feature importance metrics (for example, permutation importance) is that it contains more information that can help with understanding the global feature importance. For example, let's assume that a feature is of medium importance. Using this plot, we could see if that medium importance corresponds to the feature values having a large effect on the prediction for a few observations, but in general no effect. Or maybe it had a medium-sized effect on all predictions.

Having discussed the overall considerations, let's mention a few observations from *Figure 14.40*:

- Overall, high values of the V4 feature (the most important one) contributed to higher predictions, while lower values resulted in lower predictions (observation being less likely to be a fraudulent one).
- The overall effect of the V14 feature on the prediction was negative, but for quite a few observations with a low value of that feature, it resulted in a higher prediction.

Alternatively, we can present the same information using a bar chart. Then, we focus on the aggregate feature importance, while ignoring the insights into feature effects:

```
shap.summary_plot(shap_values, X, plot_type="bar")
```

Executing the snippet generates the following plot:

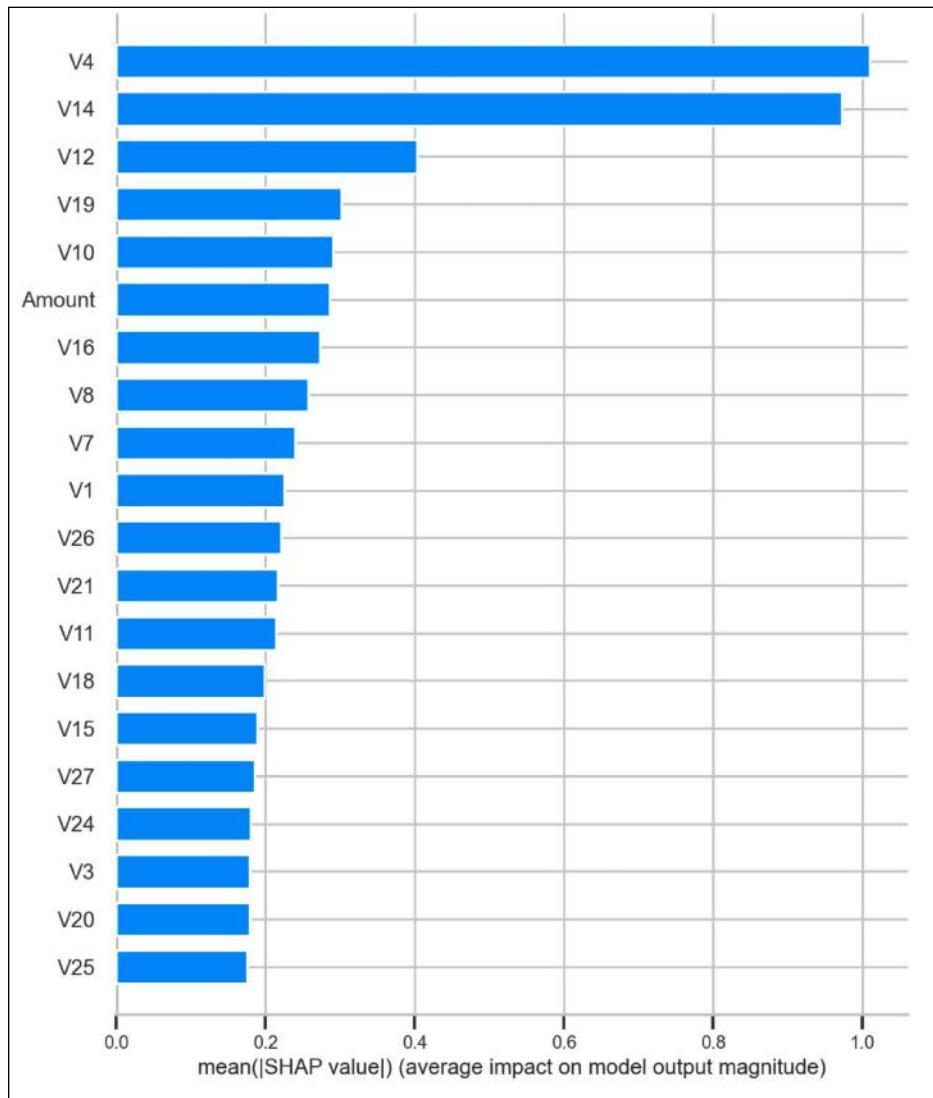


Figure 14.41: The summary plot (bar chart) calculated using the SHAP values

Naturally, the order of the features (their importance) is the same as in Figure 14.40. We could use this plot as an alternative to the permutation feature importance. However, we should then keep in mind the underlying differences. Permutation feature importance is based on the decrease in model performance (measured using a metric of choice), while SHAP is based on the magnitude of feature attributions.



We can get an even more concise representation of the summary chart using the following command: `shap.plots.bar(explainer_x)`.

9. Locate an observation belonging to the positive and negative classes:

```
negative_ind = y[y == 0].index[0]
positive_ind = y[y == 1].index[0]
```

10. Explain those observations:

```
shap.force_plot(
    explainer.expected_value,
    shap_values[negative_ind, :],
    X.iloc[negative_ind, :]
)
```

Executing the snippet generates the following plot:



Figure 14.42: An (abbreviated) force plot explaining an observation belonging to the negative class

In a nutshell, the force plot shows how features contribute to pushing the prediction from the base value (average prediction) to the actual prediction. As the plot contained much more information and it was too wide to fit the page, we only present the most relevant part. Please refer to the accompanying Jupyter notebook to inspect the full plot.

Below are some of the observations we can make based on *Figure 14.42*:

- The base value (-8.589) is the average prediction of the entire dataset.
- $f(x) = -13.37$ is the prediction of this observation.
- We can interpret the arrows as the impact of given features on the prediction. The red arrows indicate an increase in the prediction. The blue arrows indicate a decrease in the prediction. The size of the arrows corresponds to the magnitude of the feature's effect. The values by the feature names show the feature values.
- If we subtract the total length of the red arrows from the total length of the blue arrows, we will get the distance from the base value to the final prediction.
- As such, we can see that the biggest contributor to the decrease in the prediction (compared to the average prediction) was feature V14's value of -0.3112.

We then follow the same step for the positive observation:

```
shap.force_plot(
    explainer.expected_value,
    shap_values[positive_ind, :],
    X.iloc[positive_ind, :]
)
```

Executing the snippet generates the following plot:



Figure 14.43: An (abbreviated) force plot explaining an observation belonging to the positive class

Compared to Figure 14.42, we can clearly see how outbalanced the blue features (negatively impacting the prediction, labeled *lower*) are compared to the red ones (labeled *higher*). We can also see that both figures have the same base value, as this is the dataset's average predicted value.

- Create a waterfall plot for the positive observation:

```
shap.plots.waterfall(explainer(X)[positive_ind])
```

Executing the snippet generates the following plot:

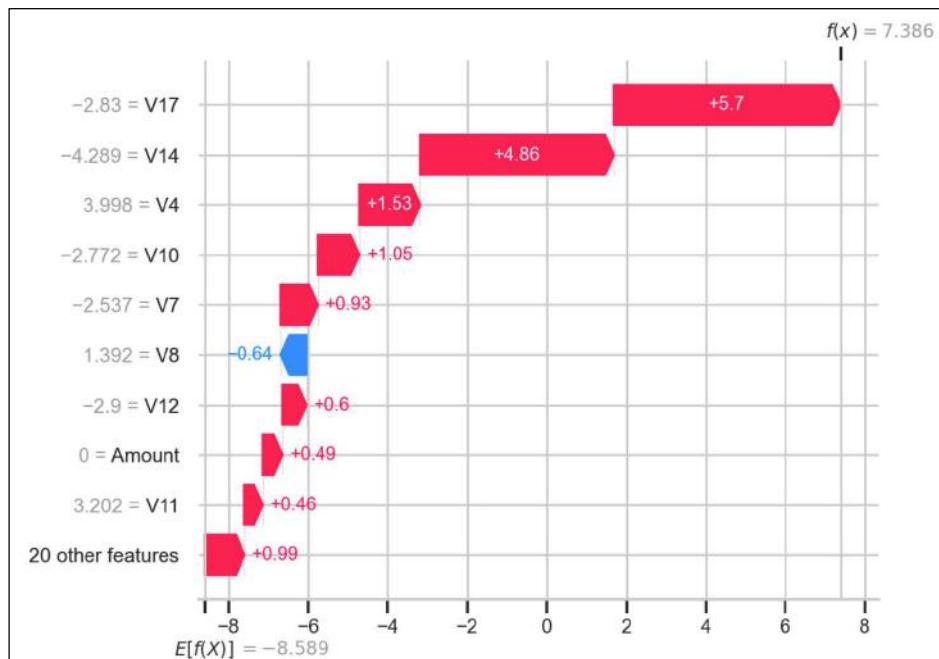


Figure 14.44: A waterfall plot explaining an observation from the positive class

Inspecting Figure 14.44 reveals many similarities to Figure 14.43, as both plots are explaining the very same observation using a slightly different visualization. Hence, most of the insights on interpreting the waterfall plot are the same as for the force plot. Some nuances include:

- The bottom of the plot starts at the baseline value (the model's average prediction). Then, each row shows the positive or negative contribution of each feature that leads to the model's final prediction for that particular observation.
- SHAP explains XGBoost classifiers in terms of their margin output. This means that the units on the x axis are log-odds units. A negative value implies probabilities lower than 0.5 that the observation was a fraudulent one.
- The least impactful features are collapsed into a joint term. We can control that using the `max_display` argument of the function.

12. Create a dependence plot of the V4 feature:

```
shap.dependence_plot("V4", shap_values, X)
```

Executing the snippet generates the following plot:

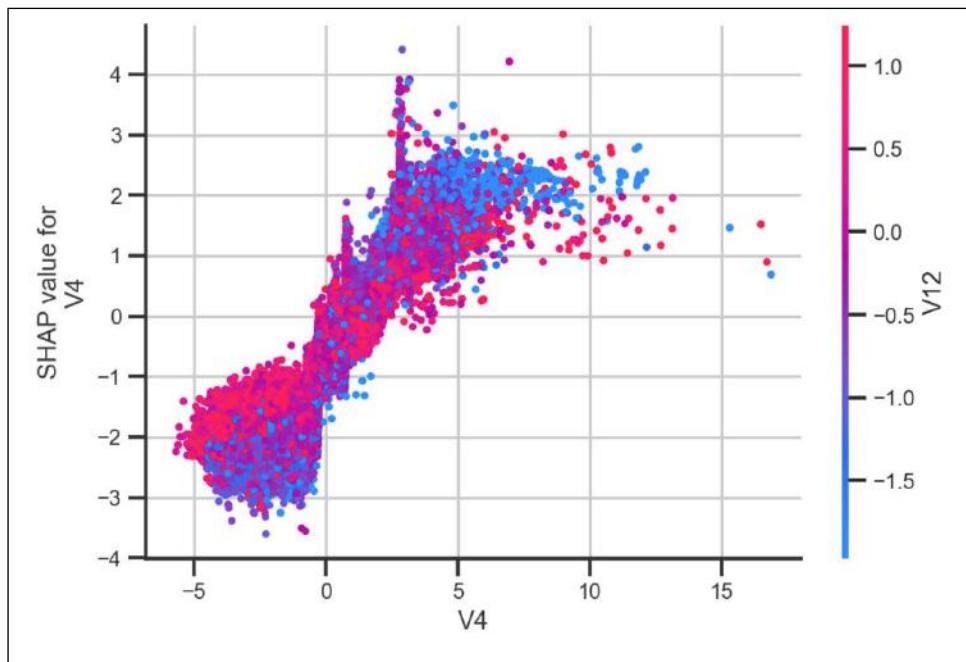


Figure 14.45: A dependence plot visualizing the dependence between the V4 and V12 features

Some things to know about a dependence plot:

- It is potentially the simplest global interpretation plot.
- This type of plot is an alternative to Partial Dependence Plots. While PDPs show the average effects, the SHAP dependence plot additionally shows the variance on the y axis. Hence it contains information about the distribution of effects.

- The plot presents the feature's value (*x* axis) vs. the SHAP value of that feature (*y* axis) across all the observations in the dataset. Each dot represents a single observation.
- Given we are explaining an XGBoost classification model, the unit of the *y* axis is the log odds of being a fraudulent case.
- The color corresponds to a second feature that may have an interaction effect with the feature we specified. It is automatically selected by the `shap` library. The documentation states that if an interaction effect is present between the two features, it will show up as a distinct vertical pattern of coloring. In other words, we should look out for clear vertical spreads between colors for the same values on the *x* axis.

To complete the analysis, we can mention a potential conclusion from *Figure 14.45*. Unfortunately, it will not be quite intuitive, as the features were anonymized.

For example, let's look at observations with the value of feature V4 around 5. For those samples, observations with lower values of feature V12 are more likely to be fraudulent than the observations with higher values of the V12 feature.

How it works...

After importing the libraries, we trained an XGBoost model to detect credit card fraud.

In *Step 3*, we plotted the ICE curves using `PartialDependenceDisplay` class. We had to provide the fitted model, the dataset (we used the training set), and the feature(s) of interest. Additionally, we provided the `subsample` argument, which specified the number of random observations from the dataset for which the ICE curves were plotted. As the dataset has over 200,000 observations, we arbitrarily chose 5,000 as a manageable number of curves to be plotted.



We have mentioned that the grid used for calculating the ICE curves most frequently consists of all the unique values available in the dataset. `scikit-learn` by default creates an equally spaced grid, covering the range between the extreme values of the feature. We can customize the grid's density using the `grid_resolution` argument.

The `from_estimator` method of `PartialDependenceDisplay` also accepts the `kind` argument, which can take the following values:

- `kind="individual"`—the method will plot the ICE curves.
- `kind="average"`—the method will display the Partial Dependence Plot.
- `kind="both"`—the method will display both the PDP and ICE curves.

In *Step 4*, we plotted the same ICE curves; however, we centered them at the origin. We did so by setting the `centered` argument to `True`. This effectively subtracts the average target value from the target vector and centers the target value at 0.

In *Step 5*, we plotted the Partial Dependence Plot, also using the `PartialDependenceDisplay.from_estimator`. As the PDP is the default value, we did not have to specify the `kind` argument. We also showed the outcome of plotting both the PDP and ICE curves in the same figure. As plotting the two-way PDP takes quite a bit of time, we sampled (without replacement) 20,000 observations from the training set.



One thing to keep in mind about `PartialDependenceDisplay` is that it treats categorical features as numeric.



Partial Dependence Plots are also available in the `pdpbox` library.

In *Step 6*, we created a more complex figure using the same functionality of `PartialDependenceDisplay`. In one figure, we plotted the individual PD plots of two features (`V4` and `V8`), and their joint (also called two-way) PD plot. To obtain the last one, we had to provide the two features of interest as a tuple. By specifying `features=["V4", "V8", ("V4", "V8")]`, we indicated that we wanted to plot two individual PD plots and then a joint one for the two features. Naturally, there is no need to plot all 3 plots in the same figure. We could have used `features=[("V4", "V8")]` to create just the joint PDP.



Another interesting angle to explore would be to overlay two Partial Dependence Plots, calculated for the same feature but using different ML models. Then we could compare if the expected impact on the prediction is similar across different models.



We have focused on plotting the ICE curves and the Partial Dependence line. However, we can also calculate those values without automatically plotting them. To do so, we can use the `partial_dependence` function. It returns a dictionary containing 3 elements: the values that create the evaluated grid, the predictions for all the points in the grid for all samples in the dataset (used for ICE curves), and the averaged values of the predictions for each point in the grid (used for the PDP).

In *Step 7*, we instantiated the `explainer` object, which is the primary class used to explain any ML/DL model using the `shap` library. To be more precise, we used the `TreeExplainer` class, as we were trying to explain an XGBoost model, that is, a tree-based model. Then, we calculated the SHAP values using the `shap_values` method of the instantiated `explainer`. To explain the model's predictions, we used the entire dataset. At this point, we could have also decided to use the training or validation/test sets.



By definition, SHAP values are very complicated to compute (an NP-hard class problem). However, thanks to the simplicity of linear models, we can read the SHAP values from a partial dependence plot. Please refer to `shap`'s documentation for more information on this topic.

In *Step 8*, we started with global explanation approaches. We generated two variants of a summary plot using the `shap.summary_plot` function. The first one was a density scatterplot of SHAP values for each of the features. It combines the overall feature importance with feature effects. We can use that information to evaluate the impact each feature has on the model's predictions (also on the observation level).

The second one was a bar chart, showing the average of the absolute SHAP values across the entire dataset. In both cases, we can use the plot to infer the feature importance calculated using SHAP values; however, the first plot provides additional information. To generate this plot, we had to additionally pass `plot_type="bar"` while calling the `shap.summary_plot` function.

After looking at the global explanations, we wanted to look into local ones. To make the analysis more interesting, we wanted to present the explanations for observations belonging to both the negative and positive classes. That is why in *Step 9* we identified the indices of such observations.

In *Step 10*, we used `shap.force_plot` to explain observation-level predictions of both observations. While calling the function, we had to provide three inputs:

- The baseline value (the average prediction for the entire dataset), which is available in the explainer object (`explainer.expected_value`)
- The SHAP values for the particular observation
- The feature values of the particular observation

In *Step 11*, we also created an observation-level plot explaining the predictions; however, we used a slightly different representation. We created a waterfall plot (using the `shap.plots.waterfall` function) to explain the positive observation. The only thing worth mentioning is that the function expects a single row of an `Explanation` object as input.

In the last step, we created a SHAP dependence plot (a global-level explanation) using the `shap.dependence_plot` function. We had to provide the feature of interest, the SHAP values, and the feature values. As the considered feature, we selected the V4 one as it was identified as the most important one by the summary plot. The second feature (V12) was determined automatically by the library.

There's more...

In this recipe, we have only provided a glimpse of the field of XAI. The field is constantly growing, as explainable methods are becoming more and more important for practitioners and businesses.

Another popular XAI technique is called LIME, which stands for **Local Interpretable Model-Agnostic Explanations**. It is an observation-level approach used for explaining the predictions of any model in an interpretable and faithful manner. To obtain the explanations, LIME locally approximates the selected hard-to-explain model with an interpretable one (such as linear models with regularization).

The interpretable models are trained on small perturbations (with additional noise) of the original observations, thus providing a good local approximation.

Treeinterpreter is another observation-level XAI method useful for explaining Random Forest models. The idea is to use the underlying trees to explain how each feature contributes to the end result. The prediction is defined as the sum of each feature's contributions and the average given by the initial node that is based on the entire training set. Using this approach, we can observe how the value of the prediction changes along the prediction path within the decision tree (after every split), combined with the information on which features caused the split, that is, a change in prediction.

Naturally, there are many more available approaches, for example:

- Ceteris-paribus profiles
- Break-down plots
- Accumulated Local Effects (ALE)
- Global surrogate models
- Counterfactual explanations
- Anchors

We recommend investigating the following Python libraries focusing on AI explainability:

- **shapash**—compiles various visualizations from SHAP/LIME as an interactive dashboard in the form of a web app.
- **explainerdashboard**—prepares a dashboard web app that explains `scikit-learn`-compatible ML models. The dashboard covers model performance, feature importance, feature contributions to individual predictions, a “what if” analysis, PDPs, SHAP values, visualization of individual decision trees, and more.
- **dalex**—the library covers various XAI methods, including variable importance, PDPs and ALE plots, breakdown and SHAP waterfall plots, and more.
- **interpret**—the InterpretML library was created by Microsoft. It covers popular explanation methods of black-box models (such as PDPs, SHAP, LIME, and so on) and allows you to train so-called glass-box models, which are interpretable. For example, `ExplainableBoostingClassifier` is designed to be fully interpretable, but at the same time provides similar accuracy to the state-of-the-art algorithms.
- **eli5**—an explainability library that provides various global and local explanations. It also covers text explanation (powered by LIME) and permutation feature importance.
- **alibi**—a library focusing on model inspection and interpretation. It covers approaches such as anchors explanations, integrated gradients, counterfactual examples, the Contrastive Explanation Method, and accumulated local effects.

See also

Additional resources are available here:

- Biecek, P., & Burzykowski, T. 2021. *Explanatory model analysis: Explore, explain and examine predictive models*. Chapman and Hall/CRC.
- Friedman, J. H. 2001. “Greedy function approximation: a gradient boosting machine.” *Annals of Statistics*: 1189–1232.
- Goldstein, A., Kapelner, A., Bleich, J., & Pitkin, E. 2015. “Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation.” *Journal of Computational and Graphical Statistics*, 24(1): 44–65.
- Hastie, T., Tibshirani, R., Friedman, J. H., & Friedman, J. H. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2: 1–758). New York: Springer.
- Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., ... & Lee, S. I. 2020. “From local explanations to global understanding with explainable AI for trees.” *Nature Machine Intelligence*, 2(1): 56–67.
- Lundberg, S. M., Erion, G. G., & Lee, S. I. 2018. “Consistent individualized feature attribution for tree ensembles.” *arXiv preprint arXiv:1802.03888*.
- Lundberg, S. M., & Lee, S. I. 2017. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, 30.
- Molnar, C. 2020. *Interpretable machine learning*. <https://christophm.github.io/interpretable-ml-book/>.
- Ribeiro, M.T., Singh, S., & Guestrin, C. 2016. “Why should I trust you?: Explaining the predictions of any classifier.” Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM.
- Saabas, A. *Interpreting random forests*. <http://blog.datadive.net/interpreting-random-forests/>.

Summary

In this chapter, we have covered a wide variety of useful concepts that can help with improving almost any ML or DL project. We started by exploring more complex classifiers (which also have their corresponding variants for regression problems), considering alternative approaches to encoding categorical features, creating stacked ensembles, and looking into possible solutions to class imbalance. We also showed how to use the Bayesian approach to hyperparameter tuning, in order to find an optimal set of hyperparameters faster than using the more popular yet uninformed grid search approaches.

We have also dived into the topic of feature importance and AI explainability. This way, we can better understand what is happening in the so-called black box models. This is crucial not only for the people working on the ML/DL project but also for any business stakeholders. Additionally, we can combine those insights with feature selection techniques to potentially further improve a model’s performance or reduce its training time.

Naturally, the data science field is constantly growing and more and more useful tools are becoming available every day. We cannot cover all of them, but below you can find a short list of libraries/tools that you might find useful in your projects:

- **DagsHub**—a platform similar to GitHub, but tailor-made for data scientists and machine learning practitioners. By integrating powerful open-source tools such as Git, DVC, MLFlow, and Label Studio and doing the DevOps heavy lifting for its users, you can easily build, manage and scale your ML project - all in one place.
- **deepchecks**—an open-source Python library for testing ML/DL models and data. We can use the library for various testing and validation needs throughout our projects; for example, we can verify our data's integrity, inspect the features' and target's distributions, confirm valid data splits, and evaluate the performance of our models.
- **DVC**—an open-source version control system for ML projects. Using **DVC (data version control)**, we can store the information about different versions of our data (be it tabular, images, or something else) and models in Git, while storing the actual data elsewhere (cloud storage like AWS, GCS, Google Drive, and so on). Using DVC, we can also create reproducible data pipelines, while storing the intermediate versions of the datasets along the way. And to make using it easier, DVC uses the same syntax as Git.
- **MLflow**—an open-source platform for managing the ML life cycle. It covers aspects such as experimentation, reproducibility, deployment, and model registry.
- **nannyML**—an open-source Python library for post-deployment data science. We can use it to identify data drift (a change in the distribution of the features between the data used for training a model and inference in production) or to estimate the model's performance in the absence of ground truth. The latter one can be especially interesting for projects in which the ground truth becomes available after a long period of time, for example, a loan default within multiple months from the moment of making the prediction.
- **pycaret**—an open-source, low-code Python library that automates a lot of the components of ML workflows. For example, we can train and tune dozens of machine learning models for a classification or regression task using as little as a few lines of code. It also contains separate modules for anomaly detection or time series forecasting.

15

Deep Learning in Finance

In recent years, we have seen many spectacular successes achieved by means of deep learning techniques. Deep neural networks have been successfully applied to tasks in which traditional machine learning algorithms could not succeed—large-scale image classification, autonomous driving, and superhuman performance when playing traditional games such as Go or classic video games (from Super Mario to StarCraft II). Almost yearly, we can observe the introduction of a new type of network that achieves state-of-the-art (SOTA) results and breaks some kind of performance record.

With the constant improvement in commercially available **Graphics Processing Units (GPUs)**, the emergence of freely available processing power involving CPUs/GPUs (Google Colab, Kaggle, and so on), and the rapid development of different frameworks, deep learning continues to gain more and more attention among researchers and practitioners who want to apply the techniques to their business cases.

In this chapter, we are going to show two possible use cases of deep learning in the financial domain—predicting credit card default (a classification task) and forecasting time series. Deep learning proves to deliver great results with sequential data such as speech, audio, and video. That is why it naturally fits into working with sequential data such as time series—both univariate and multivariate. Financial time series are known to be erratic and complex, hence the reason why it is such a challenge to model them. Deep learning approaches are especially apt for the task, as they make no assumptions about the distribution of the underlying data and can be quite robust to noise.



In the first edition of the book, we focused on the traditional NN architectures used for time series forecasting (CNN, RNN, LSTM, and GRU) and their implementation in PyTorch. In this book, we will be using more complex architectures with the help of dedicated Python libraries. Thanks to those, we do not have to recreate the logic of the NNs and we can focus on the forecasting challenges instead.

In this chapter, we present the following recipes:

- Exploring fastai's Tabular Learner
- Exploring Google's TabNet

- Time series forecasting with Amazon's DeepAR
- Time series forecasting with NeuralProphet

Exploring fastai's Tabular Learner

Deep learning is not often associated with tabular or structured data, as this kind of data comes with some possible questions:

- How should we represent features in a way that can be understood by the neural networks? In tabular data, we often deal with numerical and categorical features, so we need to correctly represent both types of inputs.
- How do we use feature interactions, both between the features themselves and the target?
- How do we effectively sample the data? Tabular datasets tend to be smaller than typical datasets used for solving computer vision or NLP problems. There is no easy way to apply augmentation, such as random cropping or rotation in the case of images. Also, there is no general large dataset with some universal properties, based on which we could easily apply transfer learning.
- How do we interpret the predictions of a neural network?

That is why practitioners tend to use traditional machine learning approaches (often based on some kind of gradient-boosted trees) to approach tasks involving structured data. However, a potential benefit of using deep learning for structured data is the fact that it requires much less feature engineering and domain knowledge.

In this recipe, we present how to successfully use deep learning for tabular data. To do so, we use the popular `fastai` library, which is built on top of PyTorch.

Some of the benefits of working with the `fastai` library are:

- It provides a selection of APIs that greatly simplify working with **Artificial Neural Networks (ANNs)**—from loading and batching the data to training the model
- It incorporates a selection of empirically tested best approaches to using deep learning for various tasks, such as image classification, NLP, and tabular data (both classification and regression problems)
- It handles the data preprocessing automatically—we just need to define which operations we want to apply

What makes `fastai` stand out is the use of **entity embedding** (or embedding layers) for categorical data. By using it, the model can learn some potentially meaningful relationships between the observations of categorical features. You can think of embeddings as latent features. For each categorical column, there is a trainable embedding matrix and each unique value has a designated vector mapped to it. Thankfully, `fastai` does all of that for us.

Using entity embedding comes with quite a few advantages. First, it reduces memory usage and speeds up the training of neural networks as compared to using one-hot encoding. Second, it maps similar values close to each other in the embedding space, which reveals the intrinsic properties of the categorical variables. Third, the technique is especially useful for datasets with many high-cardinality features, when other approaches tend to result in overfitting.

In this recipe, we apply deep learning to a classification problem based on the credit card default dataset. We have already used this dataset in *Chapter 13, Applied Machine Learning: Identifying Credit Default*.

How to do it...

Execute the following steps to train a neural network to classify defaulting customers:

1. Import the libraries:

```
from fastai.tabular.all import *
from sklearn.model_selection import train_test_split
from chapter_15_utils import performance_evaluation_report_fastai
import pandas as pd
```

2. Load the dataset from a CSV file:

```
df = pd.read_csv("../Datasets/credit_card_default.csv",
                 na_values="")
```

3. Define the target, lists of categorical/numerical features, and preprocessing steps:

```
TARGET = "default_payment_next_month"

cat_features = list(df.select_dtypes("object").columns)
num_features = list(df.select_dtypes("number").columns)
num_features.remove(TARGET)

preprocessing = [FillMissing, Categorify, Normalize]
```

4. Define the splitter used to create training and validation sets:

```
splits = RandomSplitter(valid_pct=0.2, seed=42)(range_of(df))
splits
```

Executing the snippet generates the following previews of the datasets:

```
((#24000) [27362,16258,19716,9066,1258,23042,18939,24443,4328,4976...],
 (#6000) [7542,10109,19114,5209,9270,15555,12970,10207,13694,1745...])
```

5. Create the TabularPandas dataset:

```
tabular_df = TabularPandas(
    df,
    procs=preprocessing,
    cat_names=cat_features,
    cont_names=num_features,
    y_names=TARGET,
    y_block=CategoryBlock(),
    splits=splits
)

PREVIEW_COLS = ["sex", "education", "marriage",
                 "payment_status_sep", "age_na", "limit_bal",
                 "age", "bill_statement_sep"]
tabular_df.xs.iloc[:5][PREVIEW_COLS]
```

Executing the snippet generates the following preview of the dataset:

	sex	education	marriage	payment_status_sep	age_na	limit_bal	age	bill_statement_sep
27362	2	4	3		10	1	-0.290227	-0.919919
16258	1	4	1		10	1	-0.443899	-0.266960
19716	1	1	3		2	1	2.014862	-0.158134
9066	1	2	3		3	1	-0.674408	-0.919919
1258	2	1	3		1	1	0.324464	-0.266960
								-0.692228

Figure 15.1: The preview of the encoded dataset

We printed only a small selection of columns to keep the DataFrame readable. We can observe the following:

- The categorical columns are encoded using a label encoder
- The continuous columns are normalized
- The continuous column that had missing values (age) has an extra column with an encoding indicating whether the particular value was missing before imputation

6. Define a DataLoaders object from the TabularPandas dataset:

```
data_loader = tabular_df.dataloaders(bs=64, drop_last=True)
data_loader.show_batch()
```

Executing the snippet generates the following preview of the batch:

	sex	education	marriage	payment_status_sep	payment_status_aug	payment_status_jul	payment_status_jun	payment_status_may
0	Male	Graduate school	Single	Payed duly	Payed duly	Unknown	Unknown	Unknown
1	Female	Graduate school	Single	Payed duly	Payed duly	Payed duly	Unknown	Unknown
2	Female	High school	Married	Unknown	Unknown	Unknown	Unknown	Unknown
3	Male	High school	Married	Unknown	Unknown	Unknown	Unknown	Unknown
4	Female	University	Single	Payed duly	Payed duly	Payed duly	Payed duly	Payed duly
5	Male	University	Married	Payment delayed 1 month	Payment delayed 2 months	Payment delayed 2 months	Unknown	Unknown
6	Female	University	Married	Unknown	Unknown	Unknown	Unknown	Unknown
7	Female	Graduate school	Single	Unknown	Unknown	Unknown	Unknown	Unknown
8	Female	Graduate school	Single	Unknown	Unknown	Unknown	Unknown	Unknown
9	Female	Graduate school	Single	Payed duly	Payed duly	Payed duly	Payed duly	Payed duly

Figure 15.2: The preview of a batch from the DataLoaders object

As we can see in *Figure 15.2*, the features here are in their original representation.

7. Define the metrics of choice and the tabular learner:

```
recall = Recall()
precision = Precision()
learn = tabular_learner(
    data_loader,
    [500, 200],
    metrics=[accuracy, recall, precision]
)
learn.model
```

Executing the snippet prints the schema of the model:

```
TabularModel(
    embeds): ModuleList(
        (0): Embedding(3, 3)
        (1): Embedding(5, 4)
        (2): Embedding(4, 3)
        (3): Embedding(11, 6)
        (4): Embedding(11, 6)
        (5): Embedding(11, 6)
        (6): Embedding(11, 6)
        (7): Embedding(10, 6)
        (8): Embedding(10, 6)
        (9): Embedding(3, 3)
    )
```

```
(emb_drop): Dropout(p=0.0, inplace=False)
(bn_cont): BatchNorm1d(14, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(layers): Sequential(
    (0): LinBnDrop(
        (0): Linear(in_features=63, out_features=500, bias=False)
        (1): ReLU(inplace=True)
        (2): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
    )
    (1): LinBnDrop(
        (0): Linear(in_features=500, out_features=200, bias=False)
        (1): ReLU(inplace=True)
        (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
    )
    (2): LinBnDrop(
        (0): Linear(in_features=200, out_features=2, bias=True)
    )
)
)
```

To provide an interpretation of the embeddings, `Embedding(11, 6)` means that a categorical embedding was created with 11 input values and 6 output latent features.

8. Find the suggested learning rate:

```
learn.lr_find()
```

Executing the snippet generates the following plot:

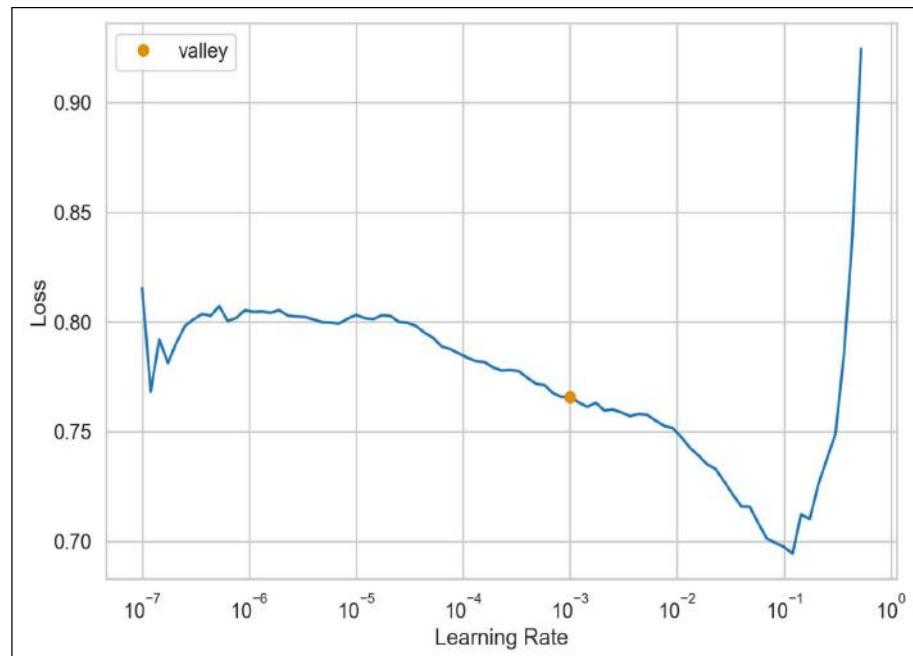


Figure 15.3: The suggested learning rate for our model

It also prints the following output with the exact value of the suggested learning rate:

```
SuggestedLRs(valley=0.001000000474974513)
```

9. Train the tabular learner:

```
learn.fit(n_epoch=25, lr=1e-3, wd=0.2)
```

While the model is training, we can observe the updates of its performance after each epoch. We present a snippet below.

epoch	train_loss	valid_loss	accuracy	recall_score	precision_score	time
0	0.448089	0.437355	0.819500	0.321596	0.655502	00:02
1	0.434199	0.440029	0.819667	0.383412	0.625000	00:01
2	0.438905	0.435580	0.823000	0.369327	0.648352	00:01
3	0.447201	0.423070	0.826667	0.366980	0.670000	00:02
4	0.440682	0.429869	0.821667	0.370110	0.640921	00:02
5	0.426624	0.424457	0.825500	0.344288	0.677966	00:01
6	0.433349	0.422457	0.825333	0.327856	0.689145	00:02
7	0.438918	0.422934	0.824000	0.343505	0.669207	00:01
8	0.431417	0.423400	0.825667	0.360720	0.668116	00:02
9	0.431647	0.429696	0.824167	0.371674	0.653370	00:01

Figure 15.4: The first 10 epochs of the Tabular learner's training

In the first 10 epochs, the losses are still a bit erratic and increase/decrease over time. The same goes for the evaluation metrics.

10. Plot the losses:

```
learn.recorder.plot_loss()
```

Executing the snippet generates the following plot:

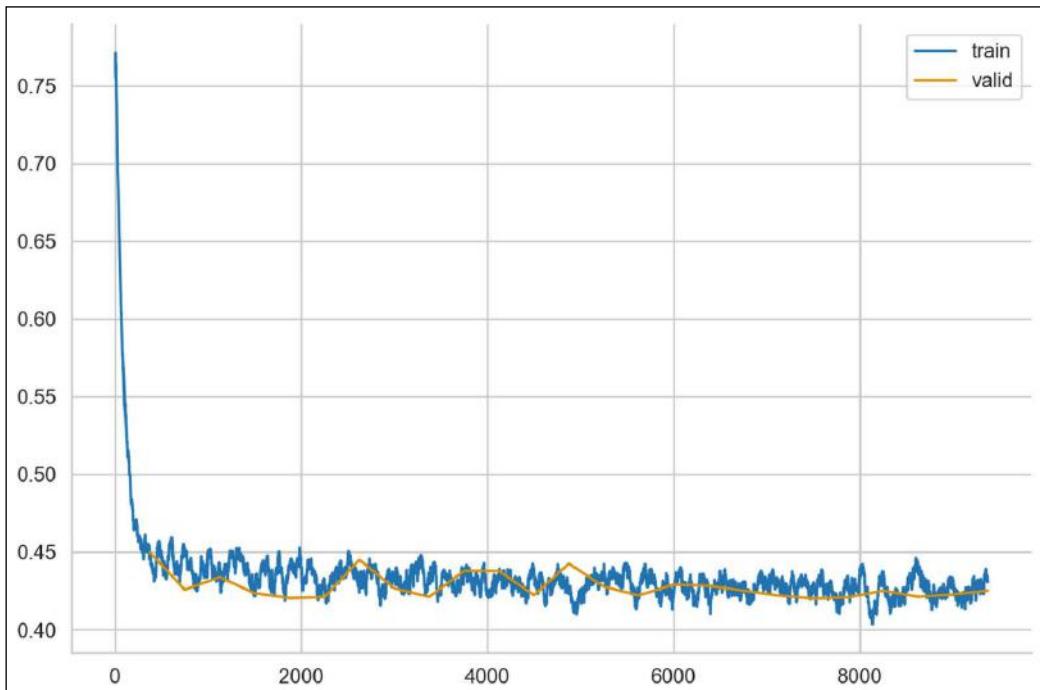


Figure 15.5: The training and validation loss over training time (batches)

We can observe that the validation loss plateaued a bit, with some bumps every now and then. It might mean that the model is a bit too complex for our data and we might want to reduce the size of the hidden layers.

11. Define the validation DataLoaders:

```
valid_data_loader = learn.dls.test_dl(df.loc[list(splits[1])])
```

12. Evaluate the performance on the validation set:

```
learn.validate(dl=valid_data_loader)
```

Executing the snippet generates the following output:

```
(#4)[0.424113571643829, 0.824833334922, 0.36228482003129, 0.66237482117310]
```

These are the metrics for the validation set: loss, accuracy, recall, and precision.

13. Get predictions for the validation set:

```
preds, y_true = learn.get_preds(dl=valid_data_loader)
```

y_true contains the actual labels from the validation set. The preds object is a tensor containing the predicted probabilities. It looks as follows:

```
tensor([[0.8092, 0.1908],
       [0.9339, 0.0661],
       [0.8631, 0.1369],
       ...,
       [0.9249, 0.0751],
       [0.8556, 0.1444],
       [0.8670, 0.1330]])
```

To get the predicted classes from it, we can use the following command:

```
preds.argmax(dim=-1)
```

14. Inspect the performance evaluation metrics:

```
perf = performance_evaluation_report_fastai(
    learn, valid_data_loader, show_plot=True
)
```

Executing the snippet generates the following plot:

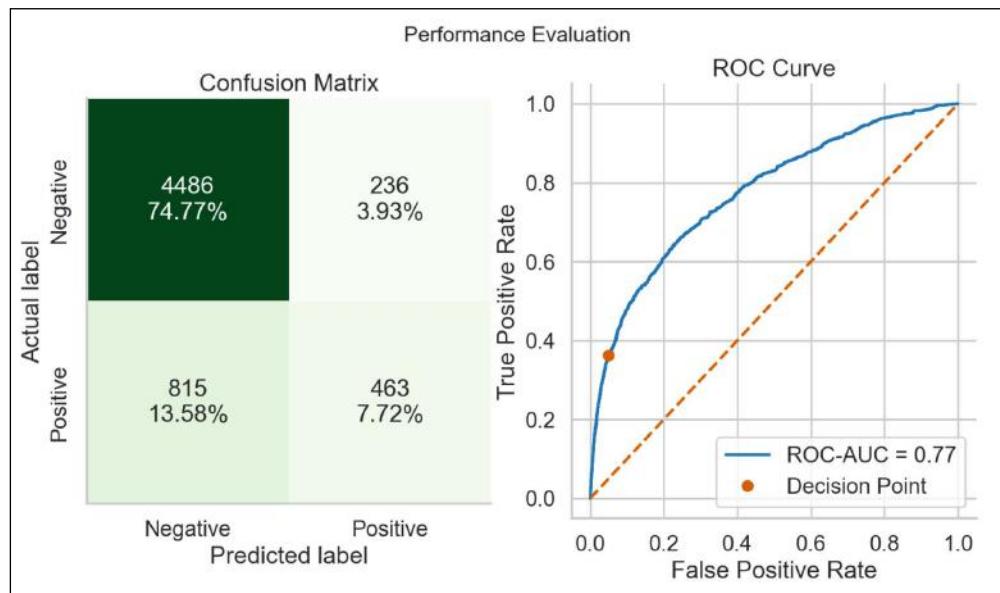


Figure 15.6: The performance evaluation of the Tabular learner's prediction of the validation set

The `perf` object is a dictionary containing various evaluation metrics. We have not presented it here for brevity, but we can also see that accuracy, precision, and recall have the same values as we saw in *Step 12*.

How it works...

In *Step 2*, we loaded the dataset into Python using the `read_csv` function. While doing so, we indicated which symbol represents the missing values.

In *Step 3*, we identified the dependent variable (the target), as well as both numerical and categorical features. To do so, we used the `select_dtypes` methods and indicated which data type we wanted to extract. We stored the features in lists. We also had to remove the dependent variable from the list containing the numerical features. Lastly, we created a list containing all the transformations we wanted to apply to the data. We selected the following:

- `FillMissing`: Missing values will be filled depending on the data type. In the case of categorical variables, missing values become a separate category. In the case of continuous features, the missing values are filled using the median of the feature's values (default approach), the mode, or with a constant value. Additionally, an extra column is added with a flag whether the value was missing or not.
- `Categorify`: Maps categorical features into their integer representation.
- `Normalize`: Features' values are transformed such that they have zero mean and unit variance. This makes training neural networks easier.

It is important to note that the same transformations will be applied to both the training and validation sets. To prevent data leakage, the transformations are based solely on the training set.

In *Step 4*, we defined a split used for creating the training and validation sets. We used the `RandomSplitter` class, which does a stratified split under the hood. We indicated we wanted to split the data using the 80-20 ratio. Additionally, after instantiating the splitter, we also had to use the `range_of` function, which returns a list containing all the indices of our `DataFrame`.

In *Step 5*, we created a `TabularPandas` dataset. It is a wrapper around a `pandas DataFrame`, which adds a few convenient utilities on top—it handles all the preprocessing and splitting. While instantiating the `TabularPandas` class, we provided the original `DataFrame`, a list containing all the preprocessing steps, the names of the target and the categorical/continuous features, and the splitter object we defined in *Step 4*. We also specified `y_block=CategoryBlock()`. We have to do so when we are working with a classification problem and the target was already encoded into a binary representation (a column of zeroes and ones). Otherwise, it might be confused with a regression problem.

We can easily convert a `TabularPandas` object into a regular `pandas DataFrame`. We can use the `xs` method to extract the features and the `ys` method to extract the target. Additionally, we can use the `cats` and `conts` methods to extract categorical and continuous features, respectively. If we use any of the four methods directly on the `TabularPandas` object, we will extract the entire dataset. Alternatively, we can use the `train` and `valid` accessors to extract only one of the sets. For example, to extract the validation set features from a `TabularPandas` object called `tabular_df`, we could use the following snippet:

```
tabular_df.valid_xs
```

In Step 6, we converted the `TabularPandas` object into a `DataLoaders` object. To do so, we used the `dataloaders` method of the `TabularPandas` dataset. Additionally, we specified a batch size of 64 and that we wanted to drop the last incomplete batch. We displayed a sample batch using the `show_batch` method.



We could have also created a `DataLoaders` object directly from a CSV file instead of converting a `pandas` `DataFrame`. To do so, we could use the `TabularDataLoaders.from_csv` functionality.

In Step 7, we defined the learner using `tabular_learner`. First, we instantiated additional metrics: precision and recall. When using `fastai`, metrics are expressed as classes (the name is spelled in uppercase) and we first need to instantiate them before passing them to the learner.

Then, we instantiated the learner. This is the place where we defined the network's architecture. We decided to use a network with two hidden layers, with 500 and 200 neurons, respectively. Choosing the network's architecture can often be considered more an art than science and may require a significant amount of trial and error. Another popular approach is to use an architecture that worked before for someone else, for example, based on academic papers, Kaggle competitions, blog articles, and so on. As for the metrics, we wanted to consider accuracy and the previously mentioned precision and recall.

As in the case of machine learning, it is crucial to prevent overfitting with neural networks. We want the networks to be able to generalize to new data. Some of the popular techniques used for tackling overfitting include the following:

- **Weight decay:** Each time the weights are updated, they are multiplied by a factor smaller than 1 (a rule of thumb is to use values between 0.01 and 0.1).
- **Dropout:** While training the NN, some activations are randomly dropped for each mini-batch. Dropout can also be used for the concatenated vector of embeddings of categorical features.
- **Batch normalization:** This technique reduces overfitting by making sure that a small number of outlying inputs does not have too much impact on the trained network.

Then, we inspected the model's architecture. In the output, we first saw the categorical embeddings and the corresponding dropout, or in this case, the lack of it. Then, in the `(layers)` section, we saw the input layer (63 input and 500 output features), followed by the `ReLU` (**R**ectified **L**inear **U**nity) activation function, and batch normalization. Potential dropout is governed in the `LinBnDrop` layer. The same steps were repeated for the second hidden layer and then the last linear layer produced the class probabilities.



`fastai` uses a rule to determine the embedding size. The rule was chosen empirically and it selects the lower of either 600, or 1.6 multiplied by the cardinality of a variable to the power of 0.56. To figure out the embedding size manually, you can use the `get_emb_sz` function. `tabular_learner` does it under the hood if the size was not specified manually.

In *Step 8*, we tried to determine the “good” learning rate. `fastai` provides a helper method, `lr_find`, which facilitates the process. It begins to train the network while increasing the learning rate—it starts with a very low one and increases to a very large one. Then, it plots the losses against the learning rates and displays the suggested value. We should aim for a value that is before the minimum value, but where the loss still improves (decreases).

In *Step 9*, we trained the neural network using the `fit` method of the learner. We’ll briefly describe the training algorithm. The entire training set is divided into **batches**. For each batch, the network is used to make predictions, which are compared to the target values and used to calculate the error. Then, the error is used to update the weights in the network. An **epoch** is a complete run through all the batches, in other words, using the entire dataset for training. In our case, we trained the network for 25 epochs. We additionally specified the learning rate and weight decay. In *Step 10*, we plotted the training and validation loss over batches.



Without going into too much detail, by default `fastai` uses the (flattened) **cross-entropy loss function** (for classification tasks) and **Adam (Adaptive Moment Estimation)** as the optimizer. The reported training and validation losses come from the loss function and the evaluation metrics (such as recall) are not used in the training procedure.

In *Step 11*, we defined a validation dataloader. To identify the indices of the validation set, we extracted them from the splitter. In the next step, we evaluated the performance of the neural network on the validation set using the `validate` method of the learner object. As input for the method, we passed the validation dataloader.

In *Step 13*, we used the `get_preds` method to obtain the validation set predictions. To obtain the predictions from the `preds` object, we had to use the `argmax` method.

Lastly, we used the slightly modified helper function (used in the previous chapters) to recover evaluation metrics such as precision and recall.

There's more...

Some noteworthy features of `fastai` for tabular datasets include:

- Using callbacks while training neural networks. Callbacks are used to insert some custom code/logic into the training loop at different times, for example, at the beginning of the epoch or at the beginning of the fitting process.
- `fastai` provides a helper function, `add_datepart`, which extracts a variety of features from columns containing dates (such as the purchase date). Some of the extracted features may include the day of the week, the day of the month, and a Boolean for the start/end of the month/quarter/year.
- We can use the `predict` method of a fitted tabular learner to predict the class directly for a single row of the source DataFrame.

- Instead of the `fit` method, we can also use the `fit_one_cycle` method. This employs the super-convergence policy. The underlying idea is to train the network with a varying learning rate. It starts at low values, increases to the specified maximum, and goes back to low values again. This approach is considered to work better than choosing a single learning rate.
- As we were working with a relatively small dataset and a simple model, we could have quite easily trained the NN on a CPU. `fastai` naturally supports using GPUs. For more information on how to use a GPU, please see `fastai`'s documentation.
- Using custom indices for training and validation sets. This feature comes in handy when we are, for example, dealing with class imbalance and want to make sure that both the training and validation sets contain a similar ratio of classes. We can use `IndexSplitter` in combination with `scikit-learn`'s `StratifiedKFold`. We show an example of the implementation in the following snippet:

```
from sklearn.model_selection import StratifiedKFold

X = df.copy()
y = X.pop(TARGET)

strat_split = StratifiedKFold(
    n_splits=5, shuffle=True, random_state=42
)
train_ind, test_ind = next(strat_split.split(X, y))
ind_splits = IndexSplitter(valid_idx=list(test_ind))(range_of(df))

tabular_df = TabularPandas(
    df,
    procs=preprocessing,
    cat_names=cat_features,
    cont_names=num_features,
    y_names=TARGET,
    y_block=CategoryBlock(),
    splits=ind_splits
)
```

See also

For more information about `fastai`, we recommend the following:

- The `fastai` course website: <https://course.fast.ai/>.
- Howard, J., & Gugger, S. 2020. *Deep Learning for Coders with fastai and PyTorch*. O'Reilly Media. <https://github.com/fastai/fastbook>.

Additional resources are available here:

- Guo, C., & Berkhahn, F. 2016. *Entity Embeddings of Categorical Variables*. arXiv preprint arXiv:1604.06737.
- Ioffe, S., & Szegedy, C. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv preprint arXiv:1502.03167.
- Krogh, A., & Hertz, J. A. 1991. “A simple weight decay can improve generalization.” In *Advances in neural information processing systems*: 9950-957.
- Ryan, M. 2020. *Deep Learning with Structured Data*. Simon and Schuster.
- Shwartz-Ziv, R., & Armon, A. 2022. “Tabular data: Deep learning is not all you need”, *Information Fusion*, 81: 84-90.
- Smith, L. N. 2018. *A disciplined approach to neural network hyperparameters: Part 1 – learning rate, batch size, momentum, and weight decay*. arXiv preprint arXiv:1803.09820.
- Smith, L. N., & Topin, N. 2019, May. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications* (1100612). International Society for Optics and Photonics.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. 2014. “Dropout: a simple way to prevent neural networks from overfitting”, *The Journal of Machine Learning Research*, 15(1): 1929-1958.

Exploring Google’s TabNet

Another possible approach to modeling tabular data using neural networks is Google’s TabNet. As TabNet is a complex model, we will not describe its architecture in depth. For that, we refer you to the original paper (mentioned in the *See also* section). Instead, we provide a high-level overview of TabNet’s main features:

- TabNet uses raw tabular data without any preprocessing.
- The optimization procedure used in TabNet is based on gradient descent.
- TabNet combines the ability of neural networks to fit very complex functions and the feature selection properties of tree-based algorithms. By using **sequential attention** to choose features at each decision step, TabNet can focus on learning from only the most useful features.
- TabNet’s architecture employs two critical building blocks: a feature transformer and an attentive transformer. The former processes the features into a more useful representation. The latter selects the most relevant features to process during the next step.
- TabNet also has another interesting component—a learnable mask of the input features. The mask should be sparse, that is, it should select a small set of features to solve the prediction task. In contrast to decision trees (and other tree-based models), the feature selection enabled by the mask allows for **soft decisions**. In practice, it means that a decision can be made on a larger range of values instead of a single threshold value.

- TabNet's feature selection is instance-wise, that is, different features can be selected for each observation (row) in the training data.
- TabNet is also quite unique as it uses a single deep learning architecture for both feature selection and reasoning.
- In contrast to the vast majority of deep learning models, TabNet is interpretable (to some extent). All of the design choices allow TabNet to offer both local and global interpretability. The local interpretability allows us to visualize the feature importances and how they are combined for a single row. The global one provides an aggregate measure of each feature's contribution to the trained model (over the entire dataset).

In this recipe, we show how to apply TabNet (its PyTorch implementation) to the same credit card default dataset we covered in the previous example.

How to do it...

Execute the following steps to train a TabNet classifier using the credit card fraud dataset:

1. Import the libraries:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import recall_score

from pytorch_tabnet.tab_model import TabNetClassifier
from pytorch_tabnet.metrics import Metric
import torch

import pandas as pd
import numpy as np
```

2. Load the dataset from a CSV file:

```
df = pd.read_csv("../Datasets/credit_card_default.csv",
                 na_values="")
```

3. Separate the target from the features and create lists with numerical/categorical features:

```
X = df.copy()
y = X.pop("default_payment_next_month")

cat_features = list(X.select_dtypes("object").columns)
num_features = list(X.select_dtypes("number").columns)
```

4. Impute missing values in the categorical features, encode them using LabelEncoder, and store the number of unique categories per feature:

```
cat_dims = {}

for col in cat_features:
    label_encoder = LabelEncoder()
    X[col] = X[col].fillna("Missing")
    X[col] = label_encoder.fit_transform(X[col].values)
    cat_dims[col] = len(label_encoder.classes_)

cat_dims
```

Executing the snippet generates the following output:

```
{'sex': 3,
 'education': 5,
 'marriage': 4,
 'payment_status_sep': 10,
 'payment_status_aug': 10,
 'payment_status_jul': 10,
 'payment_status_jun': 10,
 'payment_status_may': 9,
 'payment_status_apr': 9}
```

Based on the EDA, we would assume that the `sex` feature takes two unique values. However, as we have imputed the missing values with the `Missing` category, there are three unique possibilities.

5. Create a train/valid/test split using the 70-15-15 split:

```
# create the initial split - training and temp
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y,
    test_size=0.3,
    stratify=y,
    random_state=42
)
# create the valid and test sets
X_valid, X_test, y_valid, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.5,
    stratify=y_temp,
    random_state=42
)
```

6. Impute the missing values in the numerical features across all the sets:

```
for col in num_features:  
    imp_mean = X_train[col].mean()  
    X_train[col] = X_train[col].fillna(imp_mean)  
    X_valid[col] = X_valid[col].fillna(imp_mean)  
    X_test[col] = X_test[col].fillna(imp_mean)
```

7. Prepare lists with the indices of categorical features and the number of unique categories:

```
features = X.columns.to_list()  
cat_ind = [features.index(feat) for feat in cat_features]  
cat_dims = list(cat_dims.values())
```

8. Define a custom recall metric:

```
class Recall(Metric):  
    def __init__(self):  
        self._name = "recall"  
        self._maximize = True  
  
    def __call__(self, y_true, y_score):  
        y_pred = np.argmax(y_score, axis=1)  
        return recall_score(y_true, y_pred)
```

9. Define TabNet's parameters and instantiate the classifier:

```
tabnet_params = {  
    "cat_idxs": cat_ind,  
    "cat_dims": cat_dims,  
    "optimizer_fn": torch.optim.Adam,  
    "optimizer_params": dict(lr=2e-2),  
    "scheduler_params": {  
        "step_size": 20,  
        "gamma": 0.9  
    },  
    "scheduler_fn": torch.optim.lr_scheduler.StepLR,  
    "mask_type": "sparsemax",  
    "seed": 42,  
}  
  
tabnet = TabNetClassifier(**tabnet_params)
```

10. Train the TabNet classifier:

```
tabnet.fit(
    X_train=X_train.values,
    y_train=y_train.values,
    eval_set=[
        (X_train.values, y_train.values),
        (X_valid.values, y_valid.values)
    ],
    eval_name=["train", "valid"],
    eval_metric=["auc", Recall],
    max_epochs=200,
    patience=20,
    batch_size=1024,
    virtual_batch_size=128,
    weights=1,
)
```

Below we can see an abbreviated log from the training procedure:

```
epoch 0 | loss: 0.69867 | train_auc: 0.61461 | train_recall: 0.3789 | 
valid_auc: 0.62232 | valid_recall: 0.37286 | 0:00:01s
epoch 1 | loss: 0.62342 | train_auc: 0.70538 | train_recall: 0.51539 | 
valid_auc: 0.69053 | valid_recall: 0.48744 | 0:00:02s
epoch 2 | loss: 0.59902 | train_auc: 0.71777 | train_recall: 0.51625 | 
valid_auc: 0.71667 | valid_recall: 0.48643 | 0:00:03s
epoch 3 | loss: 0.59629 | train_auc: 0.73428 | train_recall: 0.5268 | 
valid_auc: 0.72767 | valid_recall: 0.49447 | 0:00:04s
...
epoch 42 | loss: 0.56028 | train_auc: 0.78509 | train_recall: 0.6028 | 
valid_auc: 0.76955 | valid_recall: 0.58191 | 0:00:47s
epoch 43 | loss: 0.56235 | train_auc: 0.7891 | train_recall: 0.55651 | 
valid_auc: 0.77126 | valid_recall: 0.5407 | 0:00:48s

Early stopping occurred at epoch 43 with best_epoch = 23 and best_valid_
recall = 0.6191
Best weights from best epoch are automatically used!
```

11. Prepare the history DataFrame and plot the scores over epochs:

```
history_df = pd.DataFrame(tabnet.history.history)
```

Then, we start by plotting the loss over epochs:

```
history_df[["loss"]].plot(title="Loss over epochs")
```

Executing the snippet generates the following plot:

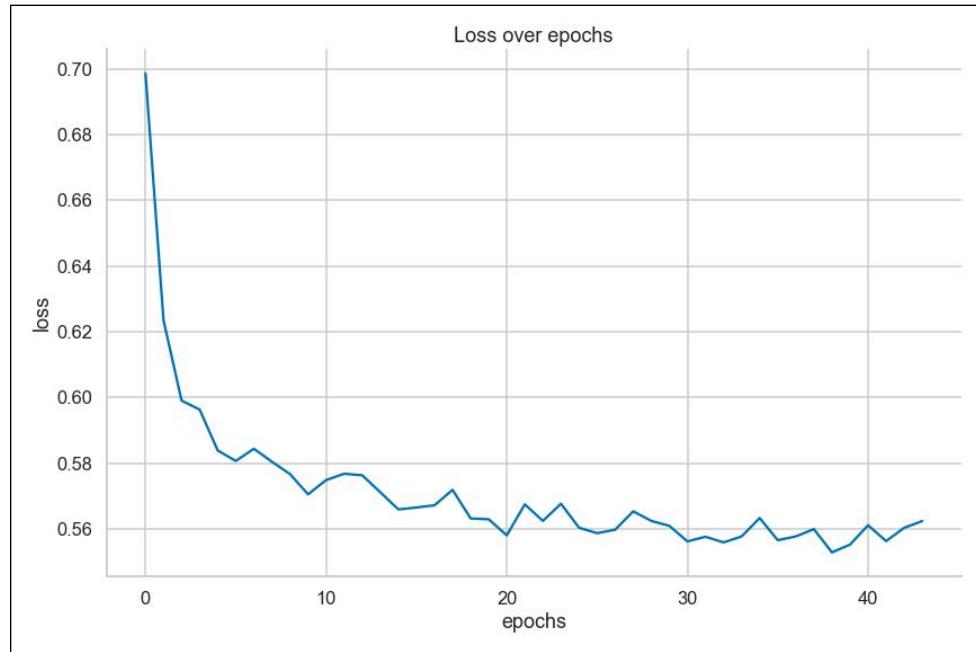


Figure 15.7: Training loss over epochs

Then, in a similar manner, we generated a plot showing the recall score over the epochs. For brevity, we have not included the code generating the plot.

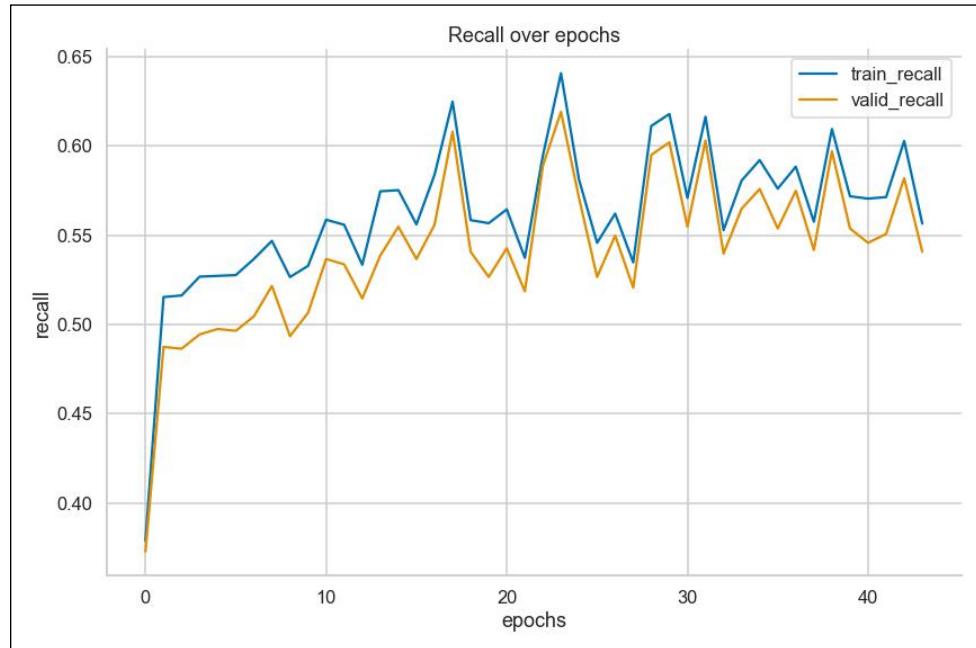


Figure 15.8: Training and validation recall over epochs

12. Create predictions for the test set and evaluate their performance:

```
y_pred = tabnet.predict(X_test.values)

print(f"Best validation score: {tabnet.best_cost:.4f}")
print(f"Test set score: {recall_score(y_test, y_pred):.4f}")
```

Executing the snippet generates the following output:

```
Best validation score: 0.6191
Test set score: 0.6275
```

As we can see, the test set performance is slightly better than the recall score calculated using the validation set.

13. Extract and plot the global feature importance:

```
tabnet_feat_imp = pd.Series(tabnet.feature_importances_,
                             index=X_train.columns)
(
    tabnet_feat_imp
    .nlargest(20)
    .sort_values()
    .plot(kind="barh",
          title="TabNet's feature importances")
)
```

Executing the snippet generates the following plot:

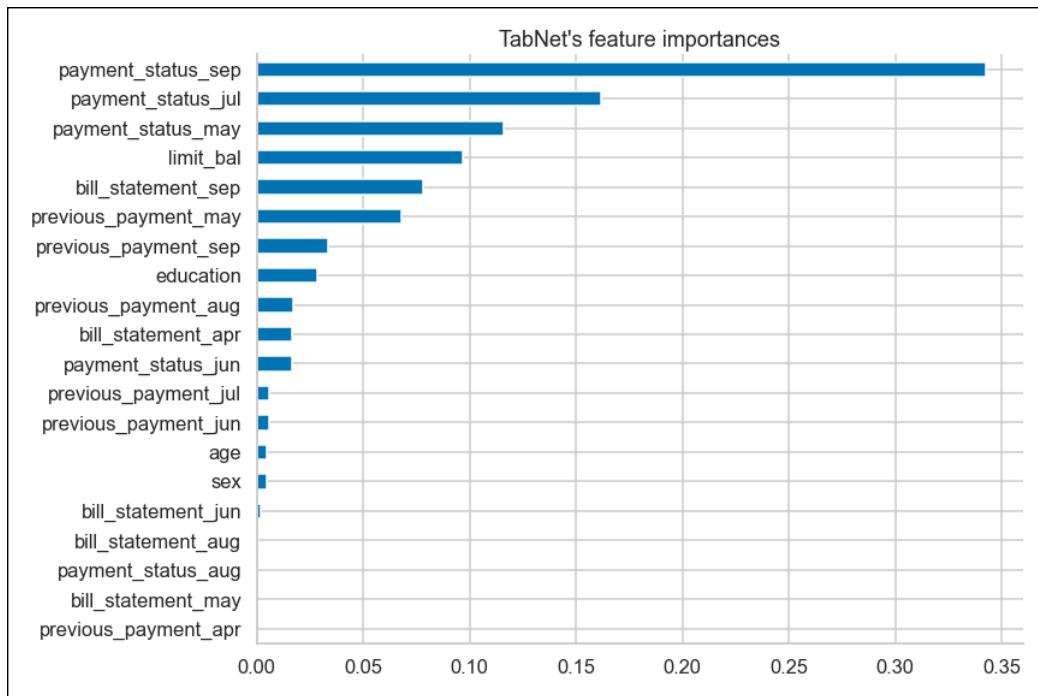


Figure 15.9: Global feature importance values extracted from the fitted TabNet classifier

According to TabNet, the most important features for predicting defaults in October were the payment statuses in September, July, and May. Another important feature was the limit balance.

Two things are worth mentioning at this point. First, the most important features are similar to the ones we identified in the *Investigating feature importance* recipe in Chapter 14, *Advanced Concepts for Machine Learning Projects*. Second, the feature importance is on a feature level, not on a feature and category level, as we could have seen while using one-hot encoding on categorical features.

How it works...

After importing the libraries, we loaded the dataset from a CSV file. Then, we separated the target from the features and extracted the names of the categorical and numerical features. We stored those as lists.

In *Step 4*, we carried out a few operations on the categorical features. First, we imputed any missing values with a new category—Missing. Then, we used `scikit-learn`'s `LabelEncoder` to encode each of the categorical columns. While doing so, we populated a dictionary containing the number of unique categories (including the newly created one for the missing values) for each of the categorical features.

In *Step 5*, we created a training/validation/test split using the `train_test_split` function. We decided to use the 70-15-15 split for the sets. As the dataset is imbalanced (the minority class is observable in approximately 22% of observations), we used stratification while splitting the data.

In *Step 6*, we imputed the missing values for the numerical features. We filled in the missing values using the average value calculated over the training set.

In *Step 7*, we prepared two lists. The first one contained the numerical indices of the categorical features, while the second one contained the number of unique categories per categorical feature. It is crucial that the lists are aligned so that the indices of the features correspond to those features' number of unique categories.

In *Step 8*, we created a custom recall metric. `pytorch-tabnet` offers a few metrics (for classification problems, those include accuracy, ROC AUC, and balanced accuracy), but we can easily define more. To create the custom metric, we did the following:

- We defined a class inheriting from the `Metric` class.
- In the `__init__` method, we defined the name of the metric (as visible in the training logs) and indicated whether the goal is to maximize the metric. That is the case for recall.
- In the `__call__` method, we calculated the value of recall using the `recall_score` function from `scikit-learn`. But first, we had to convert the array containing the predicted probabilities of each class into an object containing the predicted class. We did so using the `np.argmax` function.

In *Step 9*, we defined some of the hyperparameters of TabNet and instantiated the model. `pytorch-tabnet` offers a familiar `scikit-learn` API to train TabNet for either a classification or regression task. This way, we do not have to be familiar with PyTorch to train the model. First, we defined a dictionary containing the hyperparameters of the model.

In general, some of the hyperparameters are defined on the model level (passed to the class while instantiating it), while the other ones are defined on the fit level (passed to the model while using the `fit` method). At this point, we defined the model hyperparameters:

- The indices of the categorical features and the corresponding numbers of unique classes
- ADAM as the selected optimizer
- The learning rate scheduler
- The type of masking
- Random seed

Among all of those, the learning rate scheduler might require a bit of clarification. As per TabNet's documentation, we used a stepwise decay for the learning rate. To do so, we specified `torch.optim.lr_scheduler.StepLR` as the scheduler function. Then, we provided a few more parameters. Initially, we set the learning rate to `0.02` in the `optimizer_params`. Then, we defined the stepwise decay parameters in `scheduler_params`. We specified that after every 20 epochs, we wanted to apply the decay rate of `0.9`. In practice, it means that after 20 epochs, the learning rate will be 0.9 times `0.02`, which is equal to `0.018`. The decay then continues after every 20 epochs.

Having done so, we instantiated the `TabNetClassifier` class using the specified hyperparameters. By default, TabNet uses a cross-entropy loss function for classification problems and the MSE for regression tasks.

In *Step 10*, we trained `TabNetClassifier` using its `fit` method. We provided quite a few parameters:

- Training data
- Evaluation sets—in this case, we used both the training and validation sets so that after each epoch we see the metrics calculated over both sets
- The names of the evaluation sets
- The metrics to be used for evaluation—we used the ROC AUC and the custom recall metric defined in *Step 8*
- The maximum number of epochs
- The patience parameter, which states that if we do not observe an improvement in the evaluation metrics over `X` consecutive epochs, the training will stop and we will use the weights from the best epoch for predictions
- The batch size and the virtual batch size (used for ghost batch normalization; please see the *There's more...* section for more details)
- The `weights` parameter, which is only available for classification problems. It corresponds to sampling, which can be of great help when dealing with class imbalance. Setting it to `0` results in no sampling. Setting it to `1` turns on the sampling with the weights proportional to the inverse class occurrences. Lastly, we can provide a dictionary with custom weights for the classes.

One thing to note about TabNet's training is that the dataset we provide must be `numpy` arrays instead of `pandas` `DataFrames`. That is why we used the `values` method to extract the arrays from the `DataFrames`. The need to use `numpy` arrays is also the reason why we had to define the numeric indices of the categorical features, instead of providing a list with feature names.



Compared to many neural network architectures, TabNet uses quite large batch sizes. The original paper suggests that we can use batch sizes of up to 10% of the total number of training observations. It is also recommended that the virtual batch size is smaller than the batch size and the latter can be evenly divided into the former.

In *Step 11*, we extracted the training information from the `history` attribute of the fitted TabNet model. It contains the same information that was visible in the training log, that is, the loss, learning rate, and evaluation metrics over epochs. Then, we plotted the loss and recall over epochs.

In *Step 12*, we created the predictions using the `predict` method. Similar to the training step, we also had to provide the input features as a numpy array. As in `scikit-learn`, the `predict` method returns the predicted class, while we could use the `predict_proba` method to get the class probabilities. We also calculated the recall score over the test set using the `recall_score` function from `scikit-learn`.

In the last step, we extracted the global feature importance values. Similar to `scikit-learn` models, they are stored under the `feature_importances_` attribute of a fitted model. Then, we plotted the 20 most important features. It is worth mentioning that the global feature importance values are normalized and they sum up to 1.

There's more...

Here are a few more interesting points about TabNet and its implementation in PyTorch:

- TabNet uses **ghost batch normalization** to train large batches of data and provide better generalization at the same time. The idea behind the procedure is that we split the input batch into equal-sized sub-batches (determined by the virtual batch size parameter). Then, we apply the same batch normalization layer to those sub-batches.
- `pytorch-tabnet` allows us to apply custom data augmentation pipelines during training. Currently, the library offers using SMOTE for both classification and regression tasks.
- TabNet can be pre-trained as an unsupervised model, which can then lead to improved performance. While pre-training, certain cells are deliberately masked and the model learns the relationships between these censored cells and the adjacent columns by predicting the missing (masked) values. We can then use those weights for a supervised task. By learning about the relationships between features, the unsupervised representation learning acts as an improved encoder model for the supervised learning task. When pre-training, we can decide what percentage of features is masked.
- TabNet uses **sparsemax** as the masking function. In general, sparsemax is a non-linear normalization function with a sparser distribution than the popular softmax function. This function allows the neural network to more effectively select the important features. Additionally, the function employs sparsity regularization (its strength is determined by a hyperparameter) to penalize less sparse masks. The `pytorch-tabnet` library also contains the `EntMax` masking function.
- In the recipe, we have presented how to extract global feature importance. To extract the local ones, we can use the `explain` method of a fitted TabNet model. It returns two elements: a matrix containing the importance of each observation and feature, and the attention masks used by the model for feature selection.

See also

- Arik, S. Ö., & Pfister, T. 2021, May. Tabnet: Attentive interpretable tabular learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8): 6679-6687.
- The original repository containing TabNet's implementation described in the abovementioned paper: <https://github.com/google-research/google-research/tree/master/tabnet>.

Time series forecasting with Amazon's DeepAR

We have already covered time series analysis and forecasting in *Chapter 6, Time Series Analysis and Forecasting*, and *Chapter 7, Machine Learning-Based Approaches to Time Series Forecasting*. This time, we will have a look at an example of a deep learning approach to time series forecasting. In this recipe, we cover Amazon's DeepAR model. The model was originally developed as a tool for demand/sales forecasting at the scale of hundreds if not thousands of stock-keeping units (SKUs).

The architecture of DeepAR is beyond the scope of this book. Hence, we will only focus on some of the key characteristics of the model. Those are listed below:

- DeepAR creates a global model used for all the considered time series. It implements LSTM cells in an architecture that allows for training using hundreds or thousands of time series simultaneously. The model also uses an encoder-decoder setup, which is common in sequence-to-sequence models.
- DeepAR allows for using a set of covariates (external regressors) related to the target time series.
- The model requires minimal feature engineering. It automatically creates relevant time series features (depending on the granularity of the data, this might be the day of the month, day of the year, and so on) and it learns seasonal patterns from the provided covariates across time series.
- DeepAR offers probability forecasts based on Monte Carlo sampling—it calculates consistent quantile estimates.
- The model is able to create forecasts for time series with little historical data by learning from similar time series. This is a potential solution to the cold start problem.
- The model can use various likelihood functions.

In this recipe, we will train a DeepAR model using around 100 time series of daily stock prices from the years 2020 and 2021. Then, we will create 20-day-ahead forecasts covering the last 20 business days of 2021.

Before moving forward, we wanted to highlight that we are using time series of stock prices just for illustrative purposes. Deep learning models excel when trained on hundreds if not thousands of time series. We have selected stock prices as those are the easiest to download. As we have already mentioned, accurately forecasting stock prices, especially with a long forecast horizon, is very difficult if not simply impossible.

How to do it...

Execute the following steps to train the DeepAR model using stock prices as the input time series:

1. Import the libraries:

```
import pandas as pd
import torch
import yfinance as yf
```

```

from random import sample, seed

import pytorch_lightning as pl
from pytorch_lightning.callbacks import EarlyStopping
from pytorch_forecasting import DeepAR, TimeSeriesDataSet

```

2. Download the tickers of the S&P 500 constituents and sample 100 random tickers from the list:

```

df = pd.read_html(
    "https://en.wikipedia.org/wiki/List_of_S%26P_500_companies"
)
df = df[0]

seed(44)
sampled_tickers = sample(df["Symbol"].to_list(), 100)

```

3. Download the historical stock prices of the selected stocks:

```

raw_df = yf.download(sampled_tickers,
                     start="2020-01-01",
                     end="2021-12-31")

```

4. Keep the adjusted close price and remove the stocks with missing values:

```

df = raw_df["Adj Close"]
df = df.loc[:, ~df.isna().any()]
selected_tickers = df.columns

```

After removing the stocks that have at least one missing value in the period of interest, we are left with 98 stocks.

5. Convert the data's format from wide to long and add the time index:

```

df = df.reset_index(drop=False)

df = (
    pd.melt(df,
            id_vars=["Date"],
            value_vars=selected_tickers,
            value_name="price"
        ).rename(columns={"variable": "ticker"})
)
df["time_idx"] = df.groupby("ticker").cumcount()
df

```

Executing the snippet generates the following preview of the DataFrame:

	Date	ticker	price	time_idx
0	2019-12-31	ABC	81.503716	0
1	2020-01-02	ABC	81.561249	1
2	2020-01-03	ABC	80.535492	2
3	2020-01-06	ABC	81.714615	3
4	2020-01-07	ABC	81.129845	4
...
48980	2021-12-23	XYL	116.300835	500
48981	2021-12-27	XYL	117.082787	501
48982	2021-12-28	XYL	118.300217	502
48983	2021-12-29	XYL	118.141861	503
48984	2021-12-30	XYL	117.884506	504

Figure 15.10: The preview of the input DataFrame for the DeepAR model

- Define constants used for setting up the model's training:

```
MAX_ENCODER_LENGTH = 40
MAX_PRED_LENGTH = 20
BATCH_SIZE = 128
MAX_EPOCHS = 30
training_cutoff = df["time_idx"].max() - MAX_PRED_LENGTH
```

- Define the training and validation datasets:

```
train_set = TimeSeriesDataSet(
    df[df["time_idx"] <= training_cutoff],
    time_idx="time_idx",
    target="price",
    group_ids=["ticker"],
    time_varying_unknown_reals=["price"],
    max_encoder_length=MAX_ENCODER_LENGTH,
    max_prediction_length=MAX_PRED_LENGTH,
)

valid_set = TimeSeriesDataSet.from_dataset(
    train_set, df, min_prediction_idx=training_cutoff+1
)
```

8. Get the dataloaders from the datasets:

```
train_dataloader = train_set.to_dataloader(  
    train=True, batch_size=BATCH_SIZE  
)  
valid_dataloader = valid_set.to_dataloader(  
    train=False, batch_size=BATCH_SIZE  
)
```

9. Define the DeepAR model and find the suggested learning rate:

```
pl.seed_everything(42)  
  
deep_ar = DeepAR.from_dataset(  
    train_set,  
    learning_rate=1e-2,  
    hidden_size=30,  
    rnn_layers=4  
)  
  
trainer = pl.Trainer(gradient_clip_val=1e-1)  
res = trainer.tuner.lr_find(  
    deep_ar,  
    train_dataloaders=train_dataloader,  
    val_dataloaders=valid_dataloader,  
    min_lr=1e-5,  
    max_lr=1e0,  
    early_stop_threshold=100,  
)  
  
fig = res.plot(show=True, suggest=True)
```

Executing the snippet generates the following plot, in which the red dot indicates the suggested learning rate.

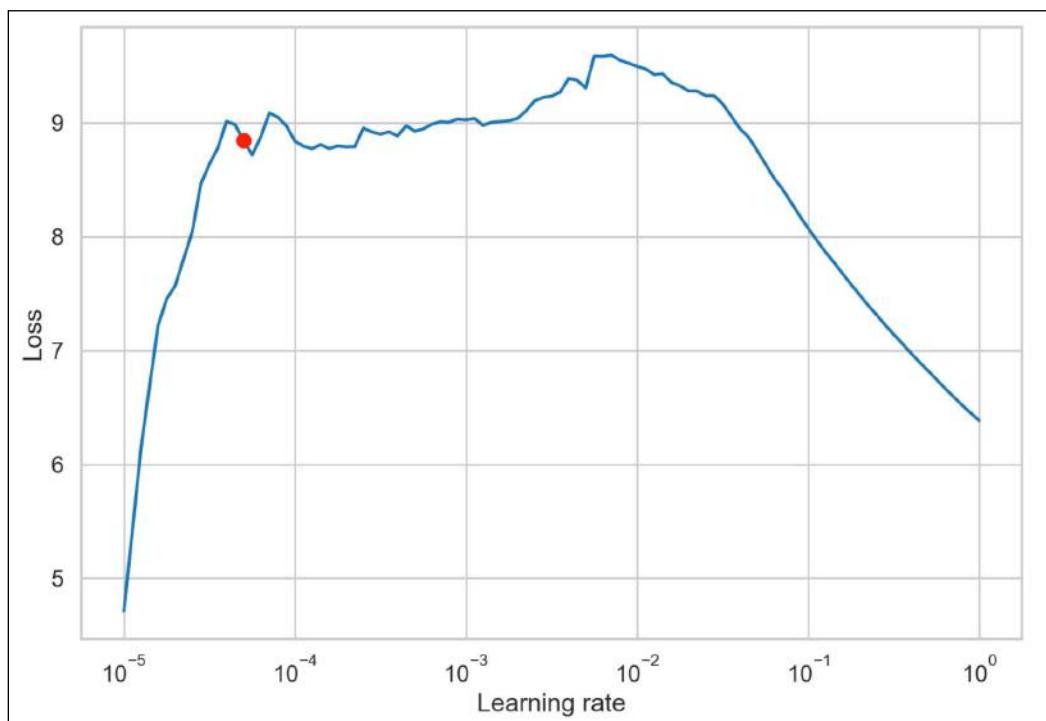


Figure 15.11: The suggested learning rate for training the DeepAR model

10. Train the DeepAR model:

```
pl.seed_everything(42)

deep_ar.hparams.learning_rate = res.suggestion()

early_stop_callback = EarlyStopping(
    monitor="val_loss",
    min_delta=1e-4,
    patience=10
)
```

```

    trainer = pl.Trainer(
        max_epochs=MAX_EPOCHS,
        gradient_clip_val=0.1,
        callbacks=[early_stop_callback]
    )

    trainer.fit(
        deep_ar,
        train_dataloaders=train_dataloader,
        val_dataloaders=valid_dataloader,
    )

```

11. Extract the best DeepAR model from a checkpoint:

```

best_model = DeepAR.load_from_checkpoint(
    trainer.checkpoint_callback.best_model_path
)

```

12. Create the predictions for the validation set and plot 5 of them:

```

raw_predictions, x = best_model.predict(
    valid_dataloader,
    mode="raw",
    return_x=True,
    n_samples=100
)

tickers = valid_set.x_to_index(x)[ "ticker" ]

for idx in range(5):
    best_model.plot_prediction(
        x, raw_predictions, idx=idx, add_loss_to_title=True
    )
    plt.suptitle(f"Ticker: {tickers.iloc[idx]}")

```

In the snippet, we generated 100 predictions and plotted 5 of them for visual inspection. For brevity, we will only show two of them. But we highly encourage inspecting more plots to better understand the model's performance.

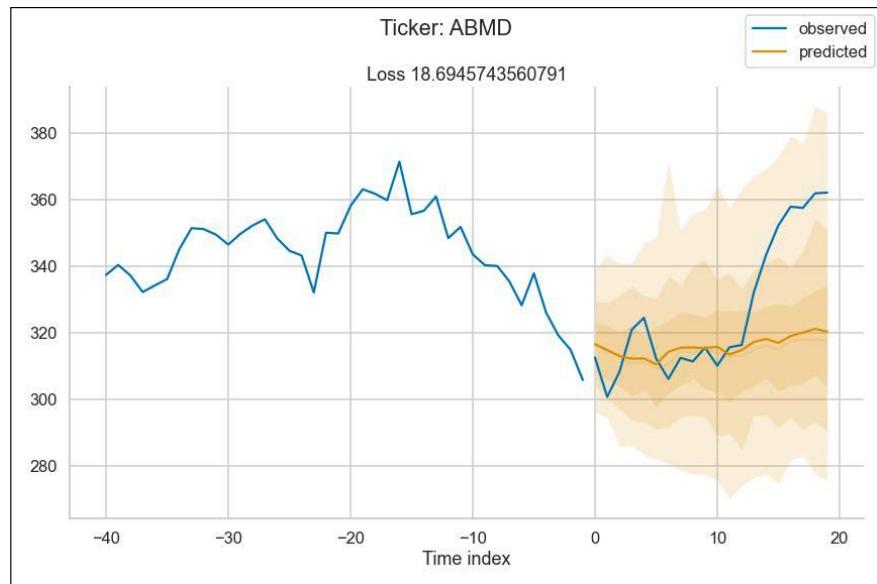


Figure 15.12: DeepAR's forecast for the ABMD stock

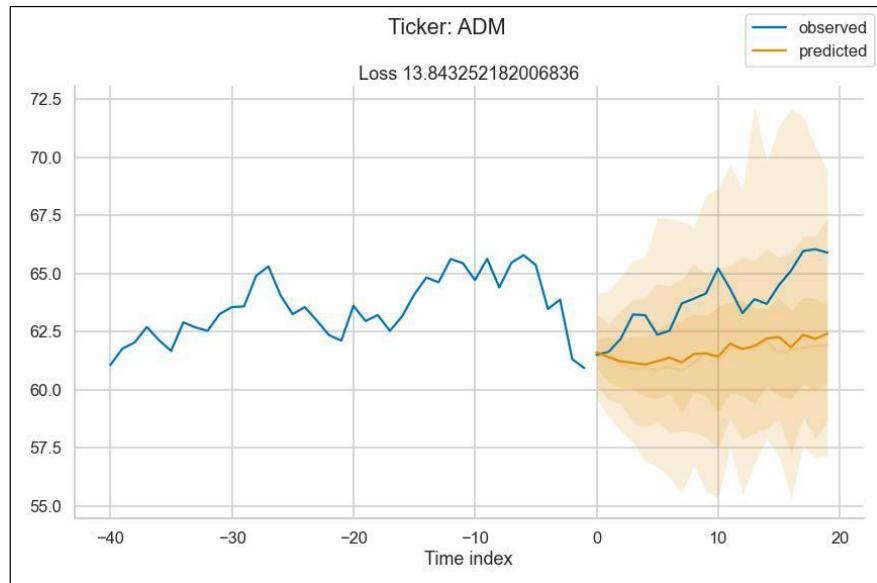


Figure 15.13: DeepAR's forecast for the ADM stock

The plots show the forecast for two stocks for the last 20 business days of 2021, together with the corresponding quantile estimates. While the forecasts do not perform that well, we can see that at the very least the actual values are within the provided quantile estimates.

We will not spend more time evaluating the performance of the model and its forecasts, as the main idea was to present how the DeepAR model works and how to use it to generate the forecasts. However, we will mention a few potential improvements. First, we could have trained for more epochs, as we did not look into the model's convergence. We have used early stopping, but it was not triggered while training. Second, we have used quite a few arbitrary values to define the network's architecture. In a real-life scenario, we should use a hyperparameter optimization routine of our choice to identify the best values for our task at hand.

How it works...

In *Step 1*, we imported the required libraries. To use the DeepAR model, we decided to use the PyTorch Forecasting library. It is a library built on top of PyTorch Lightning and allows us to easily use state-of-the-art deep learning models for time series forecasting. The models can be trained using GPUs and we can also refer to TensorBoard for inspection of the training logs.

In *Step 2*, we downloaded the list containing the constituents of the S&P 500 index. Then, we randomly sampled 100 of those and stored the results in a list. We sampled the tickers to make the training faster. It would definitely be interesting, and beneficial to the model, to repeat the exercise with all of the stocks.

In *Step 3*, we downloaded the historical stock prices from the years 2020 and 2021 using the `yfinance` library. In the next step, we had to apply further preprocessing. We only kept the adjusted close prices and we removed the stocks with any missing values.

In *Step 5*, we continued with the preprocessing. We converted the DataFrame from a wide to a long format and then added the time index. The DeepAR implementation works with an integer time index instead of dates, hence we used the `cumcount` method combined with the `groupby` method to create the time index for each of the considered stocks.

In *Step 6*, we defined some of the constants used for the training procedure, for example, the max length of the encoder step, the number of observations we wanted to forecast into the future, the max number of training epochs, and so on. We also specified which time index cuts off the training from the validation.

In *Step 7*, we defined the training and validation datasets. We did so using the `TimeSeriesDataSet` class, the responsibilities of which include:

- The handling of variable transformations
- The treatment of missing values
- Storing information about static and time-varying variables (both known and unknown in the future)
- Randomized subsampling

While defining the training dataset, we had to provide the training data (filtered using the previously defined cutoff point), the name of the columns containing the time index, the target, group IDs (in our case, these were the tickers), the encoder length, and the forecast horizon.



Each sample generated from `TimeSeriesDataSet` is a subsequence of a full-time series. Each subsequence consists of the encoder and prediction timepoints for a given time series. `TimeSeriesDataSet` creates an index defining which subsequences exist and can be sampled from.

In *Step 8*, we converted the datasets into dataloaders using the `to_dataloader` method of a `TimeSeriesDataSet`.

In *Step 9*, we defined the DeepAR model using the `from_dataset` method of the `DeepAR` class. This way, we did not have to repeat what we had already specified while creating the `TimeSeriesDataSet` objects. Additionally, we specified the learning rate, the size of the hidden layers, and the number of RNN layers. The latter two are the most important hyperparameters of the DeepAR model and they should be tuned using some HPO framework, for example, Hyperopt or Optuna. Then, we used PyTorch Lightning's `Trainer` class to find the best learning rate for our model.



By default, the DeepAR model uses the Gaussian loss function. We could use some of the alternatives, depending on the task at hand. Gaussian distribution is the preferred choice when dealing with real-valued data. We might want to use the negative-binomial likelihood for positive count data. Beta likelihood can be a good choice for data in the unit interval, while the Bernoulli likelihood is good for binary data.

In *Step 10*, we trained the DeepAR model using the identified learning rate. Additionally, we specified the early stopping callback, which stops the training if there is no significant (defined by us) improvement in the validation loss over 10 epochs.

In *Step 11*, we extracted the best model from a checkpoint. Then, we used the best model to create predictions using the `predict` method. We created predictions for 100 sequences available in the validation dataloader. We indicated that we wanted to extract the raw predictions (this option returns a dictionary with the predictions and additional information such as the corresponding quantiles, and so on) and the inputs used for generating those predictions. Then, we plotted the predictions using the `plot_prediction` method of the fitted DeepAR model.

There's more...

PyTorch Forecasting also allows us to easily train a DeepVAR model, which is the multivariate counterpart of DeepAR. Originally, Salinas *et al.* (2019) called this model VEC-LSTM.



Both DeepAR and DeepVAR are also available in Amazon's GluonTS library.

In this section, we show how to adjust the code used for training the DeepAR model to train a DeepVAR model instead:

1. Import the libraries:

```
from pytorch_forecasting.metrics import  
MultivariateNormalDistributionLoss  
import seaborn as sns  
import numpy as np
```

2. Define the dataloaders again:

```
train_set = TimeSeriesDataSet(  
    df[df["time_idx"] <= training_cutoff],  
    time_idx="time_idx",  
    target="price",  
    group_ids=["ticker"],  
    static_categoricals=["ticker"],  
    time_varying_unknown_reals=["price"],  
    max_encoder_length=MAX_ENCODER_LENGTH,  
    max_prediction_length=MAX_PRED_LENGTH,  
)  
valid_set = TimeSeriesDataSet.from_dataset(  
    train_set, df, min_prediction_idx=training_cutoff+1  
)  
  
train_dataloader = train_set.to_dataloader(  
    train=True,  
    batch_size=BATCH_SIZE,  
    batch_sampler="synchronized"  
)  
valid_dataloader = valid_set.to_dataloader(  
    train=False,  
    batch_size=BATCH_SIZE,  
    batch_sampler="synchronized"  
)
```

There are two differences in this step. First, when we created the training dataset, we also specified the `static_categoricals` argument. Because we will forecast correlations, it is important to use series characteristics such as their tickers. Second, we also had to specify `batch_sampler="synchronized"` while creating the dataloaders. Using that option ensures that samples passed to the decoder are aligned in time.

3. Define the DeepVAR model and find the learning rate:

```
pl.seed_everything(42)

deep_var = DeepAR.from_dataset(
    train_set,
    learning_rate=1e-2,
    hidden_size=30,
    rnn_layers=4,
    loss=MultivariateNormalDistributionLoss()
)

trainer = pl.Trainer(gradient_clip_val=1e-1)
res = trainer.tuner.lr_find(
    deep_var,
    train_dataloaders=train_dataloader,
    val_dataloaders=valid_dataloader,
    min_lr=1e-5,
    max_lr=1e0,
    early_stop_threshold=100,
)
```

The last difference between training DeepVAR and DeepAR models is that for the former, we use `MultivariateNormalDistributionLoss` as the loss, instead of the default `NormalDistributionLoss`.

4. Train the DeepVAR model using the selected learning rate:

```
pl.seed_everything(42)

deep_var.hparams.learning_rate = res.suggestion()

early_stop_callback = EarlyStopping(
    monitor="val_loss",
    min_delta=1e-4,
    patience=10
)

trainer = pl.Trainer(
    max_epochs=MAX_EPOCHS,
    gradient_clip_val=0.1,
    callbacks=[early_stop_callback]
)
```

```
    trainer.fit(  
        deep_var,  
        train_dataloaders=train_dataloader,  
        val_dataloaders=valid_dataloader,  
    )
```

5. Extract the best DeepVAR model from a checkpoint:

```
best_model = DeepAR.load_from_checkpoint(  
    trainer.checkpoint_callback.best_model_path  
)
```

6. Extract the correlation matrix:

```
preds = best_model.predict(valid_dataloader,  
                           mode=("raw", "prediction"),  
                           n_samples=None)  
  
cov_matrix = (  
    best_model  
    .loss  
    .map_x_to_distribution(preds)  
    .base_dist  
    .covariance_matrix  
    .mean(0)  
)  
  
cov_diag_mult = (  
    torch.diag(cov_matrix)[None] * torch.diag(cov_matrix)[None].T  
)  
corr_matrix = cov_matrix / torch.sqrt(cov_diag_mult)
```

7. Plot the correlation matrix and the distribution of the correlations:

```
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

fig, ax = plt.subplots()

cmap = sns.diverging_palette(230, 20, as_cmap=True)

sns.heatmap(
    corr_matrix, mask=mask, cmap=cmap,
    vmax=.3, center=0, square=True,
    linewidths=.5, cbar_kws={"shrink": .5}
)

ax.set_title("Correlation matrix")
```

Executing the snippet generates the following plot:

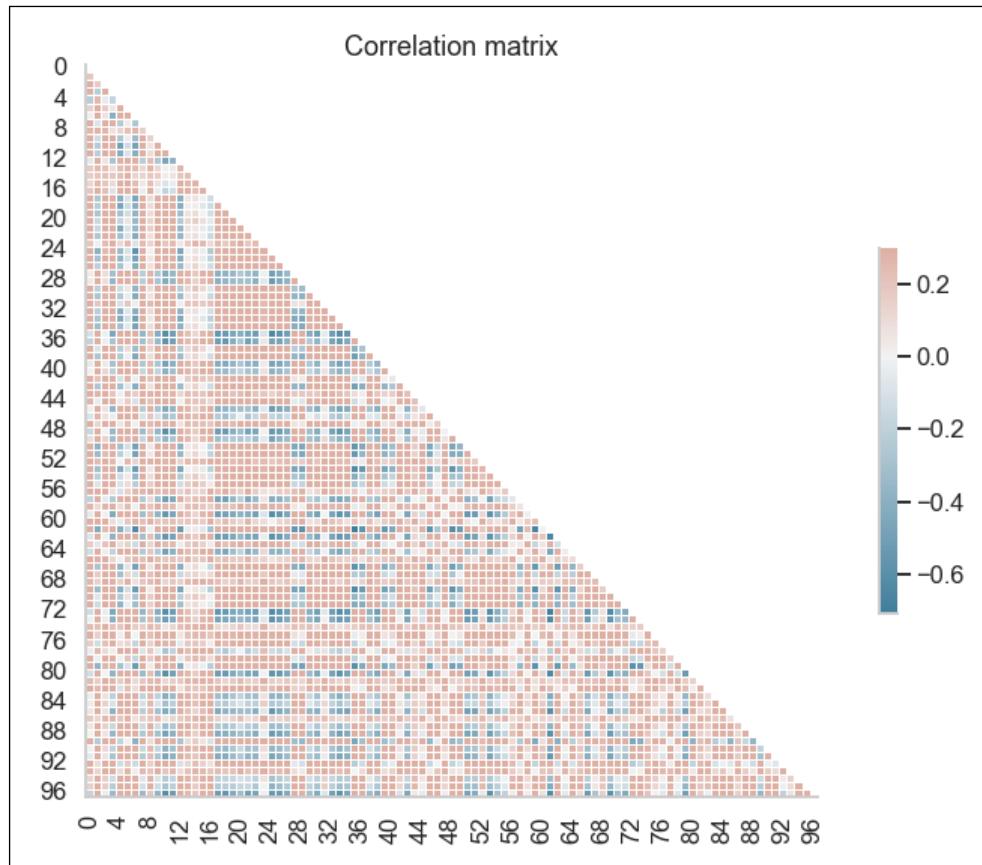


Figure 15.14: Correlation matrix extracted from DeepVAR

To get a better understanding of the distribution of correlations, we plot their histogram:

```
plt.hist(corr_matrix[corr_matrix < 1].numpy())
```

Executing the snippet generates the following plot:

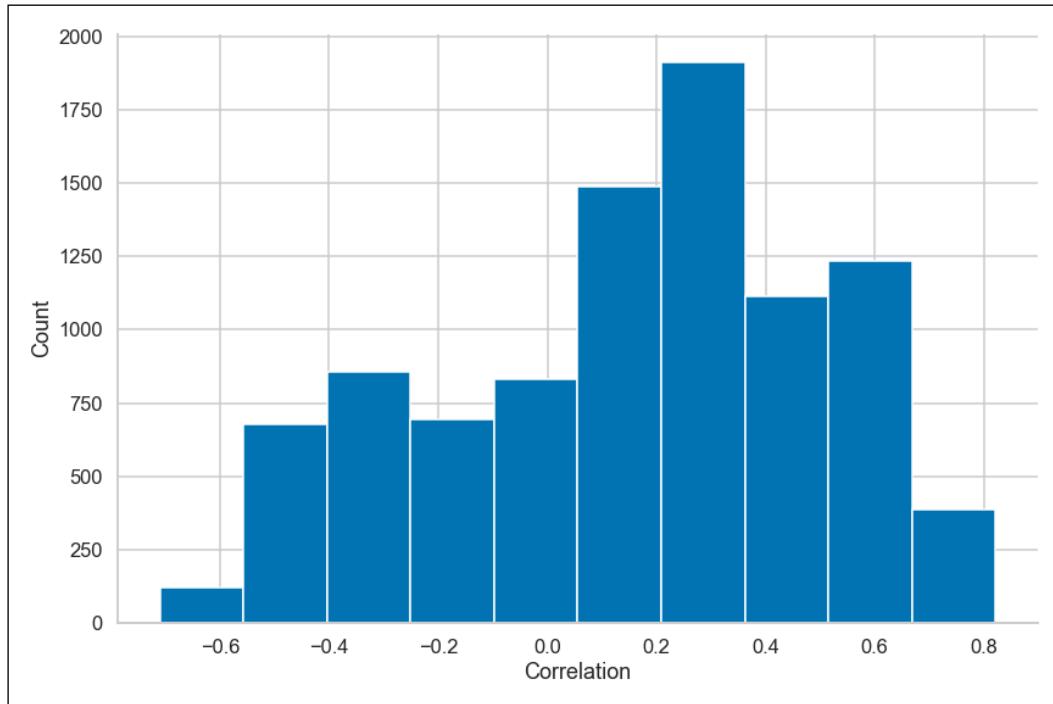


Figure 15.15: The histogram presents the distribution of the extracted correlations

While investigating the histogram, bear in mind that we have created a histogram based on the correlation matrix. This means that we have effectively counted each value twice.

See also

- Salinas, D., Flunkert, V., Gasthaus, J., & Januschowski, T. 2020. “DeepAR: Probabilistic forecasting with autoregressive recurrent networks”, *International Journal of Forecasting*, 36(3): 1181-1191.
- Salinas, D., Bohlke-Schneider, M., Callot, L., Medico, R., & Gasthaus, J. 2019. High-dimensional multivariate forecasting with low-rank Gaussian copula processes. *Advances in neural information processing systems*, 32.

Time series forecasting with NeuralProphet

In *Chapter 7, Machine Learning-Based Approaches to Time Series Forecasting*, we covered the Prophet algorithm created by Meta (formerly Facebook). In this recipe, we will look into an extension of that algorithm—NeuralProphet.

As a brief refresher, the authors of Prophet highlighted good performance, interpretability, and ease of use as the model's key advantages. The authors of NeuralProphet also had this in mind for their approach. They retained all the advantages of Prophet while adding new components that lead to improved accuracy and scalability.

The critique of the original Prophet algorithm included its rigid parametric structure (based on a generalized linear model) and the fact that it was a sort of “curve-fitter” that was not adaptive enough to fit the local patterns.



Traditionally, time series models used lagged values of the time series to predict the future value. Prophet's creators reframed time series forecasting as a curve-fitting problem and the algorithm tries to find the functional form of the trend.

In the following points, we briefly mention the most relevant additions to NeuralProphet:

- NeuralProphet introduces the autoregressive terms to the Prophet specification.
- Autoregression is included by means of the **AutoRegressive Network (AR-Net)**. AR-Net is a neural network trained to mimic the autoregressive process in a time series signal. While the inputs for the traditional AR models and AR-Net are the same, the latter is able to operate at a much larger scale than the former.
- NeuralProphet uses PyTorch as its backend, as opposed to Stan used by the Prophet algorithm. This results in faster training speed and some other benefits.
- Lagged regressors (features) are modeled using a feed-forward neural network.
- The algorithm can work with custom losses and metrics.
- The library uses regularization extensively and we are able to apply it to most of the model's components: trend, seasonality, holidays, AR terms, etc. That is especially relevant for the AR terms, as with regularization we can use more lagged values without worrying about the rapidly increasing training time.

Actually, NeuralProphet supports a few configurations of the AR terms:

- Linear AR—a single-layer neural network without bias terms or activation functions. Essentially, it regresses a particular lag onto a particular forecast step. Its simplicity makes its interpretation quite easy.
- Deep AR—in this form, the AR terms are modeled using a fully connected NN with a specified number of hidden layers and ReLU activation functions. At a cost of increased complexity, longer training time, and the loss of interpretability, this configuration often achieves higher forecast accuracy than its linear counterpart.
- Sparse AR—we can combine AR of high order (with more values at prior time steps) and the regularization term.

Each of the mentioned configurations can be applied to both the target and the covariates.

To recap, NeuralProphet is built from the following components:

- Trend
- Seasonality
- Holidays and special events
- Autoregression
- Lagged regression—lagged values of the covariates modeled internally using a feed-forward neural network
- Future regression—similar to events/holidays, these are the values of the regressors that we know in the future (either we know them as given or we have separate forecasts of those values)

In this recipe, we fit a few configurations of NeuralProphet to the time series of daily S&P 500 prices from the years 2010 to 2021. Similar to the previous recipe, we chose the time series of asset prices due to the data accessibility and its daily frequency. Trying to predict stock prices using ML/DL can be extremely hard if not impossible, so this exercise is just meant to illustrate the process of working with the NeuralProphet algorithm, rather than creating the most accurate predictions.

How to do it...

Execute the following steps to fit a few configurations of the NeuralProphet algorithm to the time series of daily S&P 500 prices:

1. Import the libraries:

```
import yfinance as yf
import pandas as pd
from neuralprophet import NeuralProphet
from neuralprophet.utils import set_random_seed
```

2. Download the historical prices of the S&P 500 index and prepare the DataFrame for modeling with NeuralProphet:

```
df = yf.download("^GSPC",
                 start="2010-01-01",
                 end="2021-12-31")
df = df[["Adj Close"]].reset_index(drop=False)
df.columns = ["ds", "y"]
```

3. Create the train/test split:

```
TEST_LENGTH = 60
df_train = df.iloc[:-TEST_LENGTH]
df_test = df.iloc[-TEST_LENGTH:]
```

4. Train the default Prophet model and plot the evaluation metrics:

```
set_random_seed(42)
model = NeuralProphet(changepoints_range=0.95)
metrics = model.fit(df_train, freq="B")

(
    metrics
    .drop(columns=["RegLoss"])
    .plot(title="Evaluation metrics during training",
          subplots=True)
)
```

Executing the snippet generates the following plot:

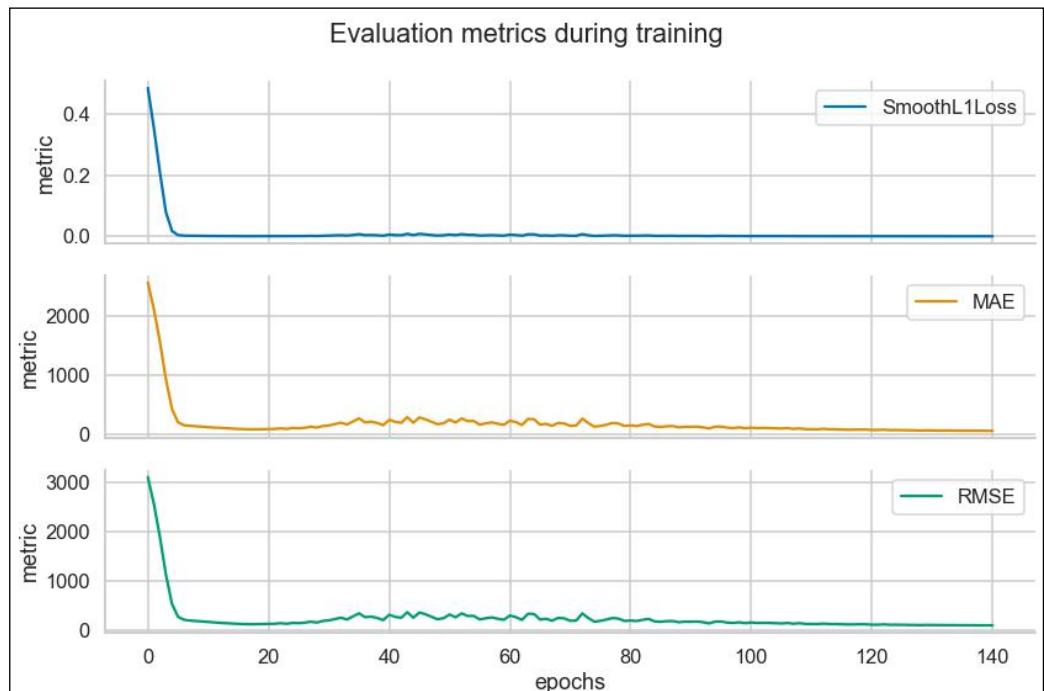


Figure 15.16: The evaluation metrics over epochs during NeuralProphet's training

5. Calculate the predictions and plot the fit:

```
pred_df = model.predict(df)

pred_df.plot(x="ds", y=["y", "yhat1"],
             title="S&P 500 - forecast vs ground truth")
```

Executing the snippet generates the following plot:

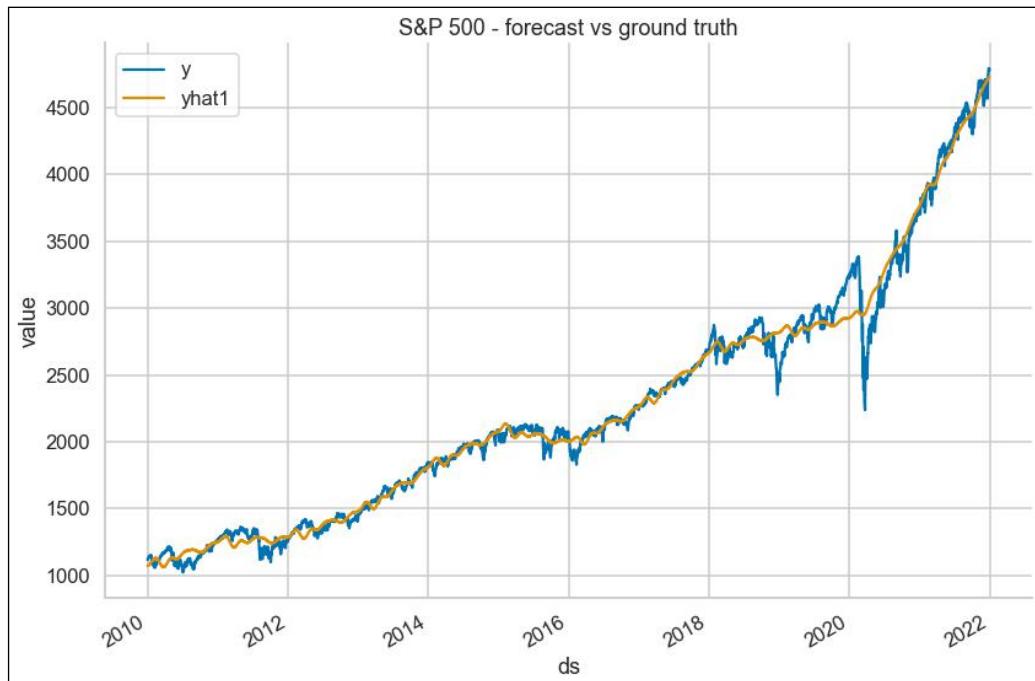


Figure 15.17: NeuralProphet's fit vs. the actual values of the entire time series

As we can see, the model's fitted line follows the overall increasing trend (it even adjusts the growth speed over time), but it misses the extreme periods and is not following the changes on the local scale.

Additionally, we can zoom in on the period corresponding to the test set:

```
(  
    pred_df  
    .iloc[-TEST_LENGTH:]  
    .plot(x="ds", y=[ "y", "yhat1"],  
          title="S&P 500 - forecast vs ground truth")  
)
```

Executing the snippet generates the following plot:

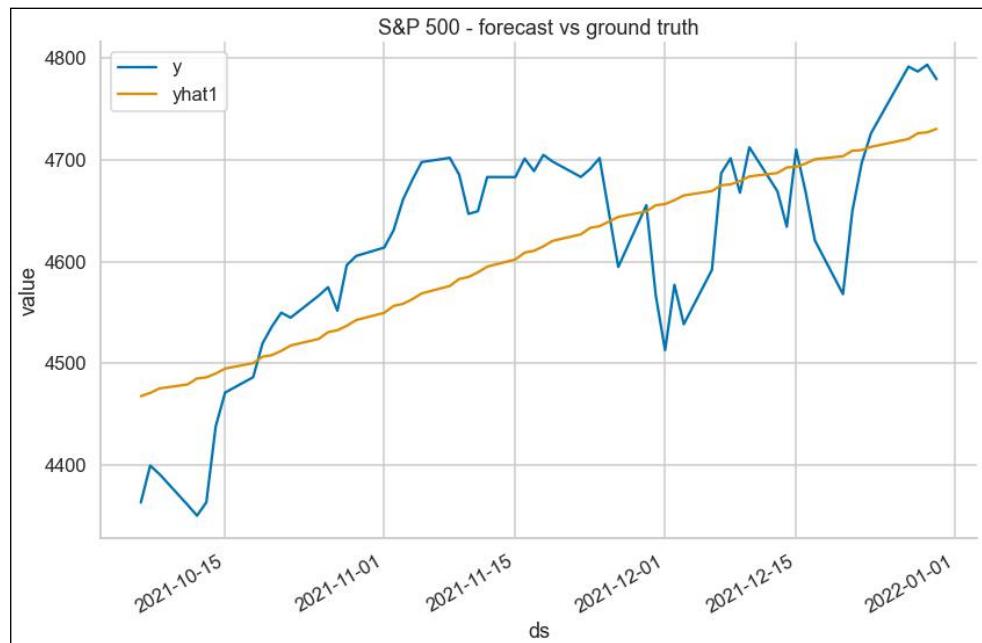


Figure 15.18: NeuralProphet's fit vs. the actual values in the test set

The conclusions from the plot are very similar to the ones we had in the case of the overall fit—the model follows the increasing trend but does not capture the local patterns.



To evaluate the performance of the test set, we can use the following command:
`model.test(df_test)`.

6. Add the AR components to NeuralProphet:

```
set_random_seed(42)
model = NeuralProphet(
    changepoints_range=0.95,
    n_lags=10,
    ar_reg=1,
)
metrics = model.fit(df_train, freq="B")

pred_df = model.predict(df)
pred_df.plot(x="ds", y=["y", "yhat1"],
             title="S&P 500 - forecast vs ground truth")
```

Executing the snippet generates the following plot:

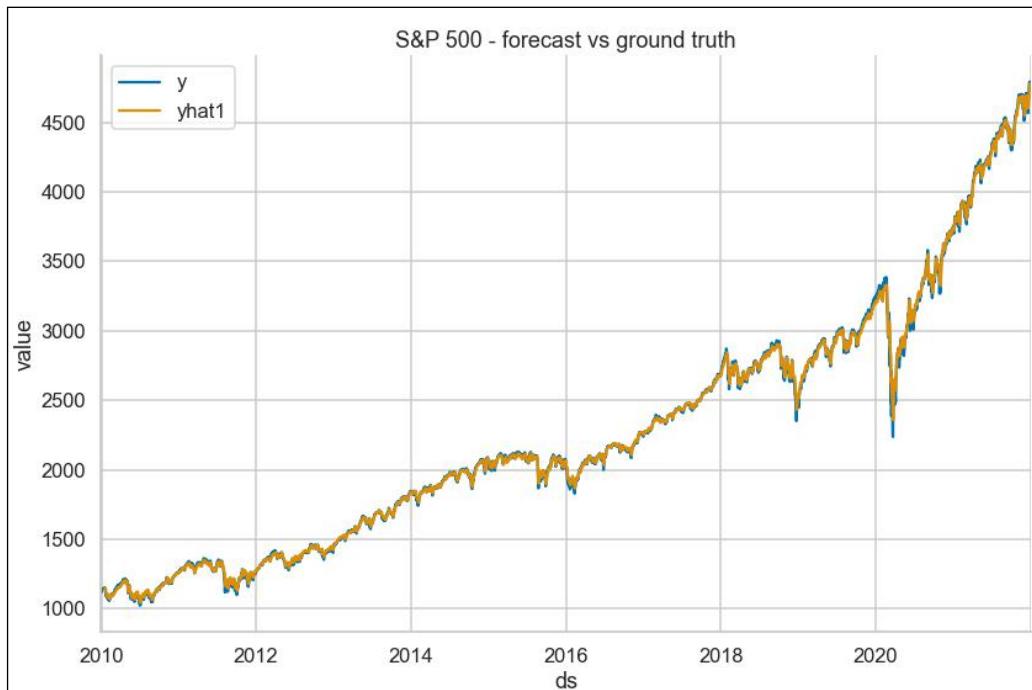


Figure 15.19: NeuralProphet's fit vs. the actual values of the entire time series

The fit looks much better than the previous one. Again, we take a closer look at the test set:

```
(  
    pred_df  
    .iloc[-TEST_LENGTH:]  
    .plot(x="ds", y=["y", "yhat1"],  
          title="S&P 500 - forecast vs ground truth")  
)
```

Executing the snippet generates the following plot:

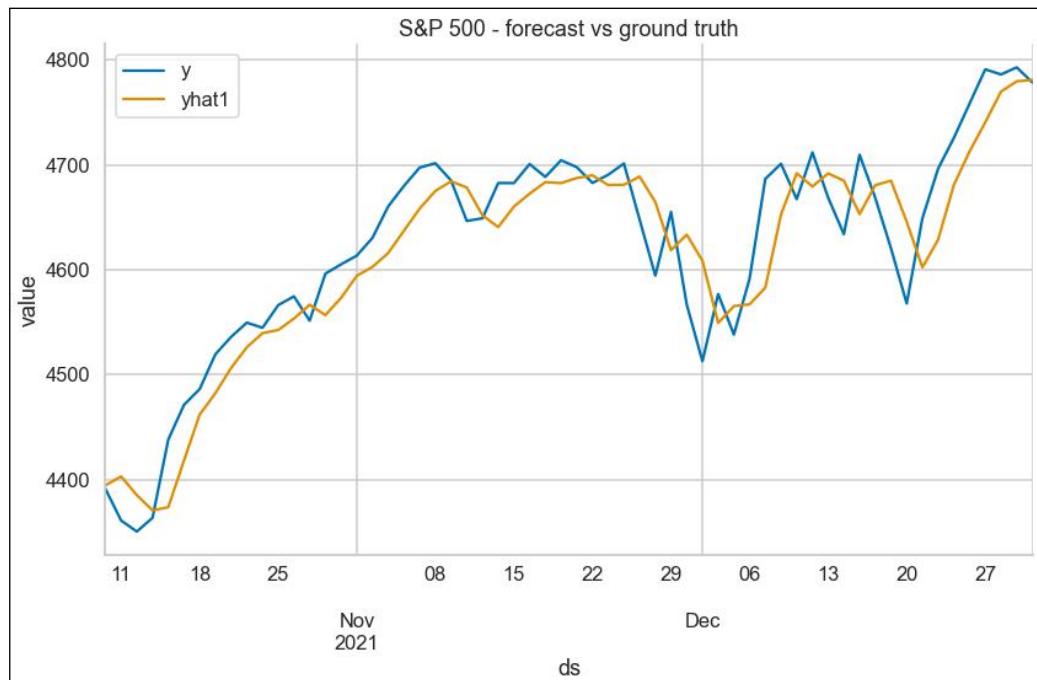


Figure 15.20: NeuralProphet's fit vs. the actual values in the test set

We can see a familiar and concerning pattern—the forecast is lagging after the original series. By that, we mean that the forecast is very similar to one of the last known values. In other words, the line of the forecast is similar to the line of the ground truth, just shifted to the right by one or multiple periods.

7. Add the AR-Net to NeuralProphet:

```
set_random_seed(42)
model = NeuralProphet(
    changepoints_range=0.95,
    n_lags=10,
    ar_reg=1,
    num_hidden_layers=3,
    d_hidden=32,
)
metrics = model.fit(df_train, freq="B")
```

```

pred_df = model.predict(df)
(
    pred_df
    .iloc[-TEST_LENGTH:]
    .plot(x="ds", y=["y", "yhat1"],
          title="S&P 500 - forecast vs ground truth")
)

```

Executing the snippet generates the following plot:

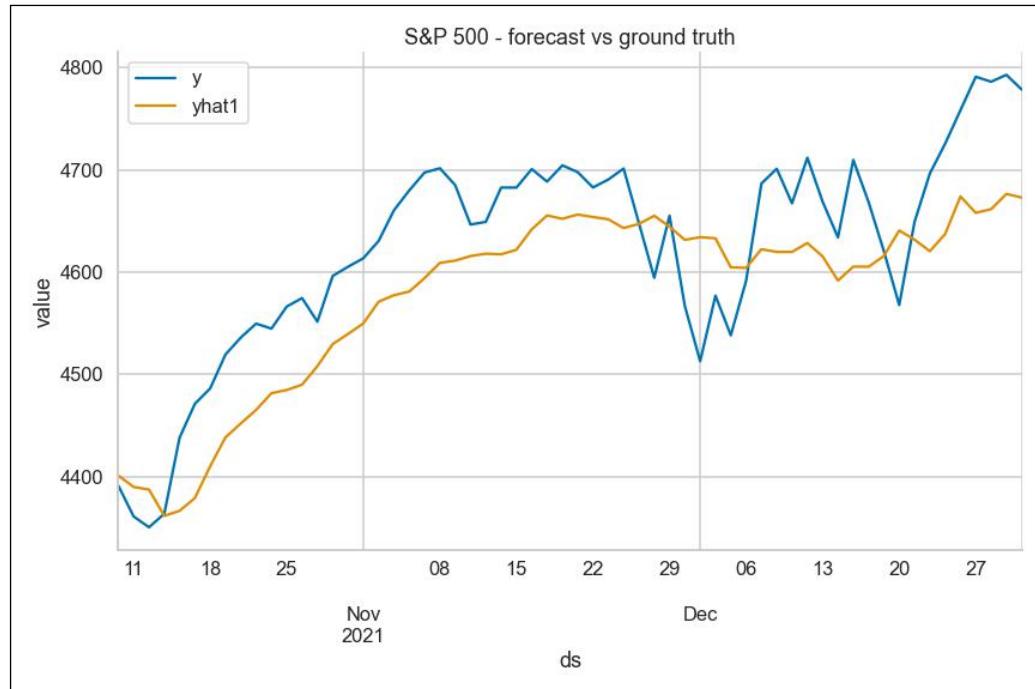


Figure 15.21: NeuralProphet's fit vs. the actual values in the test set

We can see that the plot of the forecast looks better than the one we obtained without using AR-Net. While the patterns still look shifted by a period, they are not as overfitted as in the previous case.

8. Plot the components and parameters of the model:

```
model.plot_components(model.predict(df_train))
```

Executing the snippet generates the following plots:

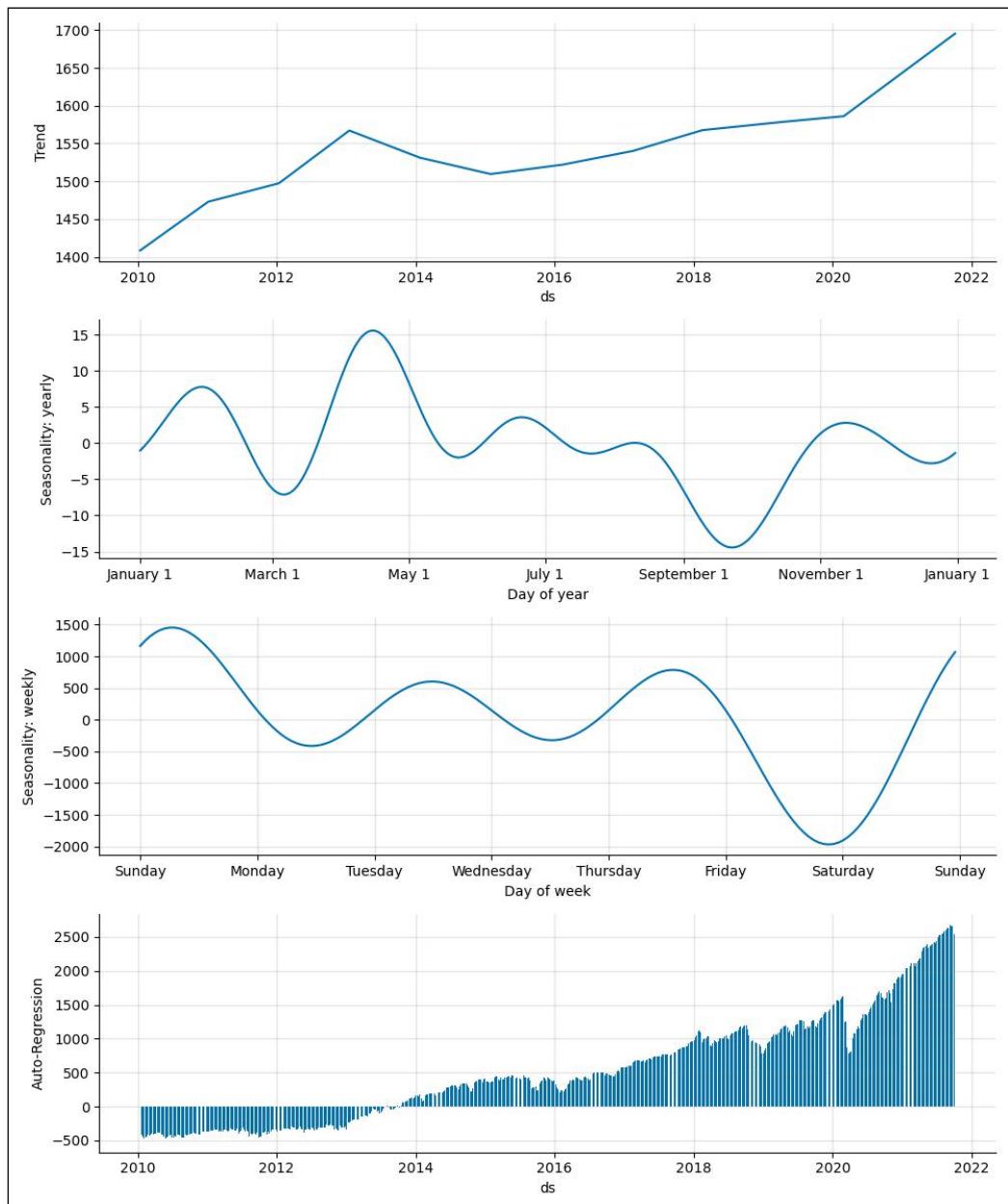


Figure 15.22: The components of the fitted NeuralProphet model (including AR-Net)

In the plots, we can see a few patterns:

- An increasing trend with a few identified changepoints.
- A seasonal peak in late April and a seasonal dip in late September and early October.
- There are no surprising patterns during the weekdays. However, it is important to remember that we should not look at the values of the weekly seasonality for Saturday and Sunday. As we are working with daily data available only on business days, the predictions should also only be made for the business days, as the intra-week seasonality will not be well estimated for the weekends.



Looking at the yearly seasonality of the stock prices can reveal some interesting patterns. One of the more famous ones is the January effect, which concerns a possible seasonal increase in stock prices in that month. Generally, it is attributed to increased buying of assets, which follows price drops in December when investors tend to sell some of their assets for tax purposes.

Then, we also plot the model's parameters:

```
model.plot_parameters()
```

Executing the snippet generates the following plots:

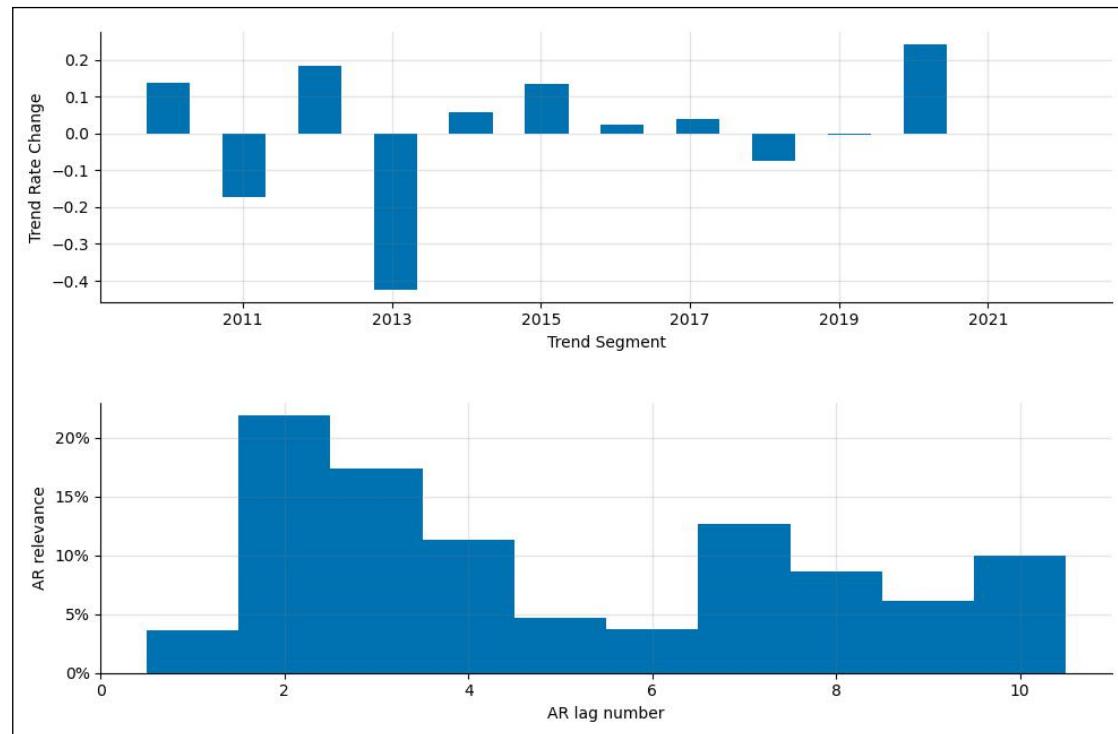


Figure 15.23: The parameters of the fitted NeuralProphet model (including AR-Net)

There is quite a lot of overlap in the components and parameters plots, hence we only focus on the new elements. First, we can look at the plot depicting the magnitudes of trend changes. We can consider it together with the plot of the trend component in *Figure 15.22*. Then, we can see how the rate of change corresponds to the trend over the years. Second, it seems that lag 2 is the most relevant of the 10 considered lags.

How it works...

After importing the libraries, we downloaded the daily prices of the S&P 500 index from the years 2010 to 2021. We only kept the adjusted close price and converted the DataFrame into a format recognized by both Prophet and NeuralProphet, that is, a DataFrame with a time column called `ds` and the target time series called `y`.

In *Step 3*, we set the test size as 60 and sliced the DataFrame into the training and test sets.



NeuralProphet also supports the use of the validation set while training the model. We can add it while calling the `fit` method.

In *Step 4*, we instantiated the almost default NeuralProphet model. The only hyperparameter we tweaked was `changepoints_range`. We increased the value from the default of 0.9 to 0.95. It means that the model can identify the changepoints in the first 95% of data. The rest is left untouched in order to ensure a consistent final trend. We increased the default value as we will be focusing on relatively short-term predictions.

In *Step 5*, we calculated the predictions using the `predict` method and the entire time series as input. This way, we obtained the fitted values (in-sample fit) and the out-of-sample predictions for the test set. At this point, we could have also used the `make_future_dataframe` method, which is familiar from the original Prophet library.

In *Step 6*, we added the linear AR terms. We specified the number of lags to consider using the `n_lags` argument. Additionally, we added the regularization of the AR terms by setting `ar_reg` to 1. We could have specified the learning rate. However, when we do not provide a value, the library uses the learning rate range test to find the best value.



When setting the regularization of the AR terms (this applies to all regularization in the library), a value of zero results in no regularization. Small values (for example, in the range of 0.001 to 1) result in weak regularization. In the case of the AR terms, this would mean that there will be more non-zero AR coefficients. Large values (for example, in the range of 1 to 100) will significantly limit the number of non-zero coefficients.

In *Step 7*, we extended the use of the AR terms from linear AR to AR-Net. We kept the other hyperparameters the same as in *Step 6*, but we specified how many hidden layers to use (`num_hidden_layers`) and what their size is (`d_hidden`).

In the last step, we plotted NeuralProphet's components using the `plot_components` method and the model's parameters using the `plot_parameters` method.

There's more...

We have just covered the basics of using NeuralProphet. In this section, we mention a few more features of the library.

Adding holidays and special events

One of the very popular features of the original Prophet algorithm that is also available in NeuralProphet is the possibility to easily add holidays and special dates. For example, when working in retail, we could add sports events (such as world championships, or the Super Bowl) or Black Friday, which is not an official holiday. In the following snippet, we add the US holidays to our model based on AR-Net:

```
set_random_seed(42)
model = NeuralProphet(
    changepoints_range=0.95,
    n_lags=10,
    ar_reg=1,
    num_hidden_layers=3,
    d_hidden=32,
)
model = model.add_country_holidays(
    "US", lower_window=-1, upper_window=1
)
metrics = model.fit(df_train, freq="B")
```

Additionally, we specify that the holidays also affect the surrounding days, that is, one day before and after the holiday. This functionality could be especially important if we consider lead-ups and draw-downs after certain dates. For example, in retail, we might want to specify a period leading up to Christmas, as that is the time when people usually buy gifts.

By inspecting the components plot, we can see the impact of the holidays over time.

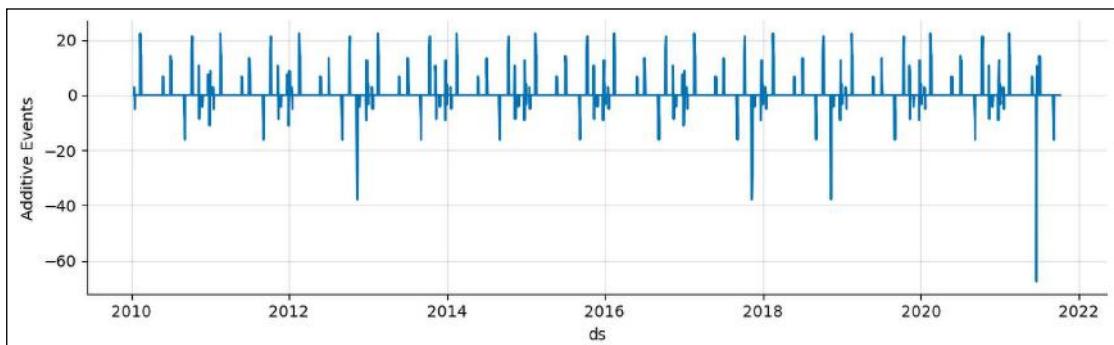


Figure 15.24: The holidays component of the fitted NeuralProphet

Additionally, we can inspect the parameters plot to gain more insights into the impact of the particular holidays (and the days around them).

In this case, we have added all US holidays at once. As a result, all the holidays also have the same range of surrounding days (one before and one after). However, we could manually create a Data-Frame with custom holidays and specify the number of surrounding days on the specific event level, instead of globally.

Next-step forecast vs. multi-step forecast

There are two approaches to forecasting multiple steps into the future using NeuralProphet:

- We can recursively create one-step ahead forecasts. The process looks as follows: we predict a step ahead, add the predicted value to the data, and then forecast the next step. We repeat the procedure until we reach the desired forecast horizon.
- We can directly forecast multiple steps ahead.

By default, NeuralProphet will use the first approach. However, we can use the second one by specifying the `n_forecasts` hyperparameter of the `NeuralProphet` class:

```
model = NeuralProphet(
    n_lags=10,
    n_forecasts=10,
    ar_reg=1,
    learning_rate=0.01
)
metrics = model.fit(df_train, freq="B")
pred_df = model.predict(df)
pred_df.tail()
```

Below we display only a part of the resulting DataFrame.

	ds	y	yhat1	residual1	yhat2	residual2	yhat3	residual3
3126	2021-12-24	4758.489990	4650.537109	-107.952881	4658.835938	-99.654053	4628.593262	-129.896729
3127	2021-12-27	4791.189941	4678.010742	-113.179199	4684.404785	-106.785156	4667.214355	-123.975586
3128	2021-12-28	4786.350098	4733.727051	-52.623047	4726.821777	-59.52832	4719.004395	-67.345703
3129	2021-12-29	4793.060059	4743.061035	-49.999023	4752.594238	-40.46582	4732.162598	-60.897461
3130	2021-12-30	4778.729980	4739.231934	-39.498047	4743.820801	-34.90918	4716.883789	-61.846191

Figure 15.25: Preview of the DataFrame containing 10-step-ahead forecasts

This time, the DataFrame will contain 10 predictions for each row: $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_{10}$. To learn how to interpret the table, we can look at the last row presented in *Figure 15.25*. The \hat{y}_2 value corresponds to the prediction for 2021-12-30, made 2 days prior to that date. So the number after \hat{y} indicates the age of the prediction (in this case, expressed in days).

Alternatively, we can shift this around. By specifying `raw=True` while calling the `predict` method, we obtain predictions made on the row's date, instead of a prediction for that date:

```
pred_df = model.predict(df, raw=True, decompose=False)
pred_df.tail()
```

Executing the snippet generated the following preview of the DataFrame:

	ds	step0	step1	step2	step3	step4
3116	2021-12-24	4650.540039	4684.399902	4719.000000	4711.359863	4727.819824
3117	2021-12-27	4678.009766	4726.819824	4732.160156	4736.580078	4733.509766
3118	2021-12-28	4733.729980	4752.589844	4716.879883	4745.649902	4740.490234
3119	2021-12-29	4743.060059	4743.819824	4702.350098	4733.750000	4747.859863
3120	2021-12-30	4739.229980	4752.520020	4730.939941	4760.689941	4776.040039

Figure 15.26: Preview of the DataFrame containing the first 5 of the 10-step-ahead forecasts

We can easily track some forecasts in both tables to see how the tables' structures differ.

When plotting a multi-step-ahead forecast, we will see multiple lines—each originating from a different date of the forecast:

```
pred_df = model.predict(df_test)
model.plot(pred_df)
ax = plt.gca()
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
ax.set_title("10-day ahead multi-step forecast")
```

Executing the snippet generates the following plot:

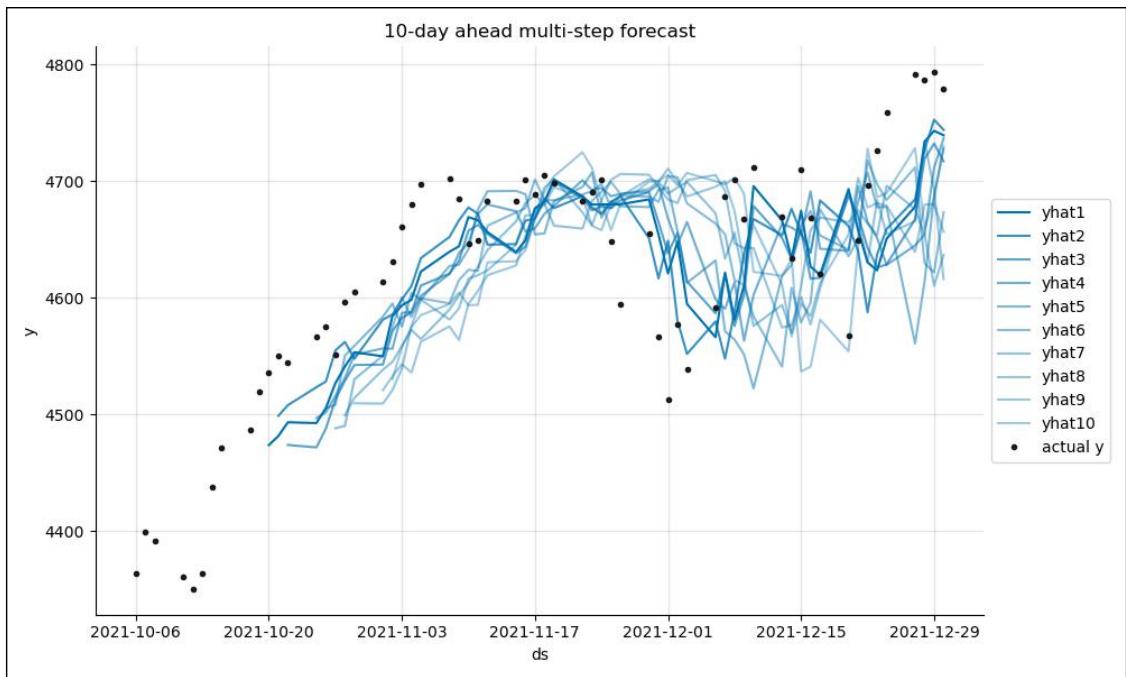


Figure 15.27: 10-day-ahead multi-step forecast

The plot is quite hard to read due to the overlapping lines. We can highlight the forecast made for a certain step using the `highlight_nth_step_ahead_of_each_forecast` method. The following snippet illustrates how to do it:

```
model = model.highlight_nth_step_ahead_of_each_forecast(1)
model.plot(pred_df)
ax = plt.gca()
ax.set_title("Step 1 of the 10-day ahead multi-step forecast")
```

Executing the snippet generates the following plot:

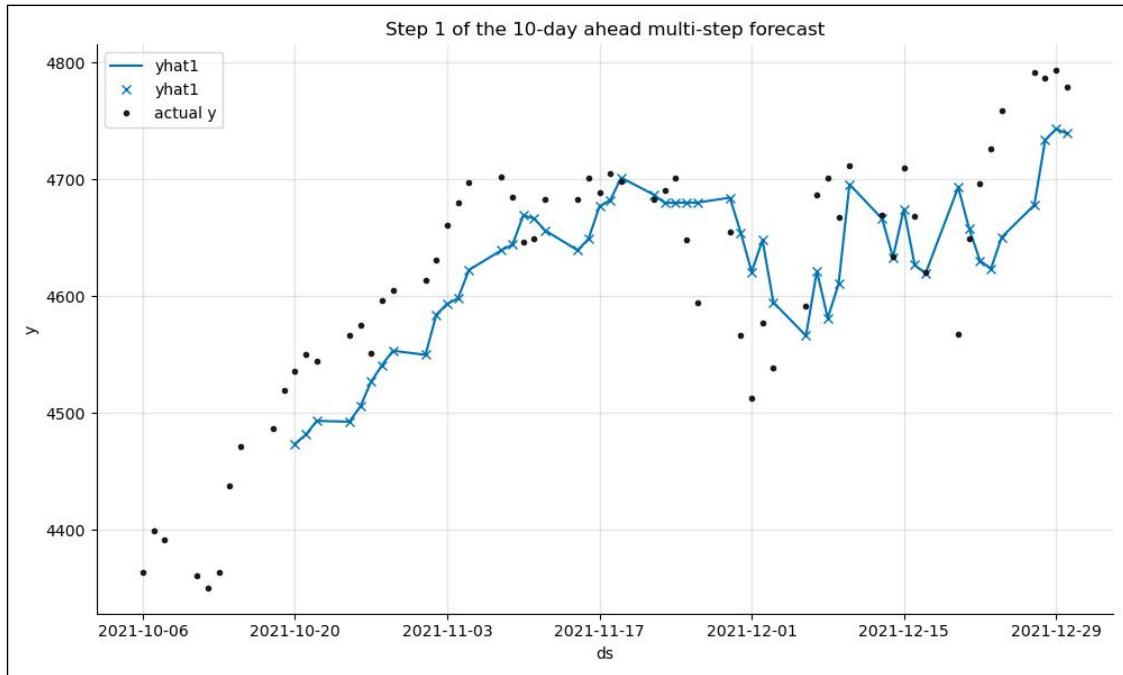


Figure 15.28: Step 1 of the 10-day multi-step forecast

After analyzing Figure 15.28, we can conclude that the model is still struggling with the predictions and the forecasted values are very close to the last known values.

Other features

NeuralProphet also contains some other interesting features, including:

- Extensive cross-validation and benchmarking functionalities
- The components of the model such as holidays/events, seasonality, or future regressors do not need to be additive; they can also be multiplicative
- The default loss function is Huber loss, but we can change it to any of the other popular loss functions

See also

- Triebe, O., Laptev, N., & Rajagopal, R. 2019. *Ar-net: A simple autoregressive neural network for time-series*. arXiv preprint arXiv:1911.12436.
- Triebe, O., Hewamalage, H., Pilyugina, P., Laptev, N., Bergmeir, C., & Rajagopal, R. 2021. *Neuralprophet: Explainable forecasting at scale*. arXiv preprint arXiv:2111.15397.

Summary

In this chapter, we explored how we can use deep learning for both tabular and time series data. Instead of building the neural networks from scratch, we used modern Python libraries which handled most of the heavy lifting for us.

As we have already mentioned, deep learning is a rapidly developing field with new neural network architectures being published daily. Hence, it is difficult to scratch even just the tip of the iceberg in a single chapter. That is why we will now point you toward some of the popular and influential approaches/libraries that you might want to explore on your own.

Tabular data

Below we list some relevant papers and Python libraries that will definitely be good starting points for further exploration of the topic of using deep learning with tabular data.

Further reading:

- Huang, X., Khetan, A., Cvitkovic, M., & Karnin, Z. 2020. *Tabtransformer: Tabular data modeling using contextual embeddings*. arXiv preprint arXiv:2012.06678.
- Popov, S., Morozov, S., & Babenko, A. 2019. *Neural oblivious decision ensembles for deep learning on tabular data*. arXiv preprint arXiv:1909.06312.

Libraries:

- `pytorch_tabular`—this library offers a framework for using deep learning models for tabular data. It provides models such as TabNet, TabTransformer, FT Transformer, and a feed-forward network with category embedding.
- `pytorch_widedeep`—a library based on Google’s Wide and Deep algorithm. It not only allows us to use deep learning with tabular data but also facilitates the combination of text and images with corresponding tabular data.

Time series

In this chapter, we have covered two deep learning-based approaches to time series forecasting—DeepAR and NeuralProphet. We highly recommend also looking into the following resources on analyzing and forecasting time series.

Further reading:

- Chen, Y., Kang, Y., Chen, Y., & Wang, Z. (2020). “Probabilistic forecasting with temporal convolutional neural network”, *Neurocomputing*, 399: 491-501.
- Gallicchio, C., Micheli, A., & Pedrelli, L. 2018. “Design of deep echo state networks”, *Neural Networks*, 108: 33-47.
- Kazemi, S. M., Goel, R., Eghbali, S., Ramanan, J., Sahota, J., Thakur, S., ... & Brubaker, M. 2019. *Time2vec: Learning a vector representation of time*. arXiv preprint arXiv:1907.05321.

- Lea, C., Flynn, M. D., Vidal, R., Reiter, A., & Hager, G. D. 2017. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 156-165.
- Lim, B., Arik, S. Ö., Loeff, N., & Pfister, T. 2021. “Temporal fusion transformers for interpretable multi-horizon time series forecasting”, *International Journal of Forecasting*, 37(4): 1748-1764.
- Oreshkin, B. N., Carpov, D., Chapados, N., & Bengio, Y. 2019. *N-BEATS: Neural basis expansion analysis for interpretable time series forecasting*. arXiv preprint arXiv:1905.10437.

Libraries:

- `tsai`—this is a deep learning library built on top of PyTorch and `fastai`. It focuses on various time series-related tasks, including classification, regression, forecasting, and imputation. Aside from already traditional approaches such as LSTMs or GRUs, it implements a selection of state-of-the-art architectures such as ResNet, InceptionTime, TabTransformer, and Rocket.
- `gluonts`—a Python library for probabilistic time series modeling using deep learning. It contains models such as DeepAR, DeepVAR, N-BEATS, Temporal Fusion Transformer, WaveNet, and many more.
- `darts`—a versatile library for time series forecasting using a variety of methods, from statistical models such as ARIMA to deep neural networks. It contains implementations of models such as N-BEATS, Temporal Fusion Transformer, and temporal convolutional neural networks.

Other domains

In this chapter, we have focused on showing the applications of deep learning in tabular data and time series forecasting. However, there are many more use cases and recent developments. For example, FinBERT is a pre-trained NLP model used to analyze the sentiment of financial texts, such as earnings call transcripts.

On the other hand, we can use the recent developments in generative adversarial networks to generate synthetic data for our models. Below, we mention some interesting starting points for further exploration of the field of deep learning in a financial context.

Further reading:

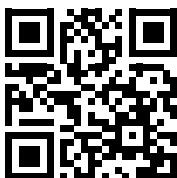
- Araci, D. 2019. *Finbert: Financial sentiment analysis with pre-trained language models*. arXiv preprint arXiv:1908.10063.
- Cao, J., Chen, J., Hull, J., & Poulos, Z. 2021. “Deep hedging of derivatives using reinforcement learning”, *The Journal of Financial Data Science*, 3(1): 10-27.
- Xie, J., Girshick, R., & Farhadi, A. 2016, June. Unsupervised deep embedding for clustering analysis. In *International conference on machine learning*, 478-487. PMLR.
- Yoon, J., Jarrett, D., & Van der Schaar, M. 2019. Time-series generative adversarial networks. *Advances in neural information processing systems*, 32.

Libraries:

- `tensortrade`—offers a reinforcement learning framework for training, evaluating, and deploying trading agents.
- `FinRL`—an ecosystem consisting of various applications of reinforcement learning in the financial context. It covers state-of-the-art algorithms, financial applications such as crypto trading or high-frequency trading, and more.
- `ydata-synthetic`—a library useful for generating synthetic tabular and time series data with the use of state-of-the-art generative models, for example, TimeGAN.
- `sdv`—the name stands for Synthetic Data Vault and it is, as the name suggests, another library useful for generating synthetic data. It covers tabular, relational, and time series data.
- `transformers`—this is a Python library that allows us to access a range of pre-trained transformer models (for example, FinBERT). The company behind the library is called Hugging Face, and it offers a platform that enables its users to build, train, and deploy ML/DL models.
- `autogluon`—this library offers AutoML for tabular data, as well as text and images. It contains various state-of-the-art ML and DL models.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



<https://packt.link/ips2H>



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

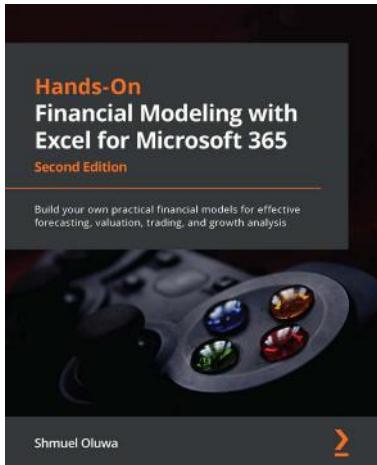
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

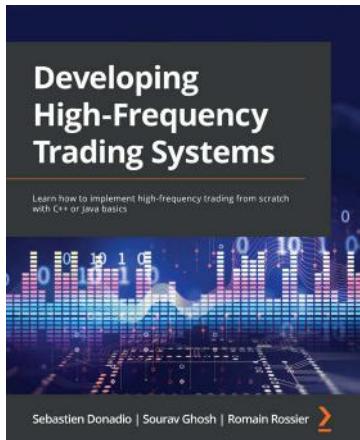


Hands-On Financial Modeling with Excel for Microsoft 365 – Second Edition

Shmuel Oluwa

ISBN: 978-1-80323-114-3

- Identify the growth drivers derived from processing historical data in Excel
- Use discounted cash flow (DCF) for efficient investment analysis
- Prepare detailed asset and debt schedule models in Excel
- Calculate profitability ratios using various profit parameters
- Obtain and transform data using Power Query
- Dive into capital budgeting techniques
- Apply a Monte Carlo simulation to derive key assumptions for your financial model
- Build a financial model by projecting balance sheets and profit and loss



Developing High-Frequency Trading Systems

Sebastien Donadio

Sourav Ghosh

Romain Rossier

ISBN: 978-1-80324-281-1

- Understand the architecture of high-frequency trading systems
- Boost system performance to achieve the lowest possible latency
- Leverage the power of Python programming, C++, and Java to build your trading systems
- Bypass your kernel and optimize your operating system
- Use static analysis to improve code development
- Use C++ templates and Java multithreading for ultra-low latency
- Apply your knowledge to cryptocurrency trading

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Python for Finance Cookbook - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

3 Month T-bill

using, from FRED database 281, 282

13 Week T-bill

using 280, 281

A

absolute percent error (APE) 261

accuracy paradox 568

Adaptive Moment Estimation (Adam) 656

Adaptive Synthetic Sampling (ADSYN) 564

drawbacks 564

additive model 144

Akaike Information Criterion (AIC) 156, 190

allowable leverage 402-405

alpha 5, 377

Alpha Vantage

data, obtaining from 16-20

reference link 21

altair

reference link 69

American options

pricing, with Least Squares Monte Carlo 353-356

pricing, with QuantLib 357-361

analysis of variance (ANOVA) 610

anchored walk-forward validation 207

antithetic variates 346

ARCH effect 104

ARCH model 306

stock returns' volatility, modeling with 306-311

area under the ROC curve (AUC) 512

ARIMA (Autoregressive Integrated Moving Average) 305

ARIMA class models

reference links 189

time series, modeling with 176-189

Artificial Neural Networks (ANNs) 500, 646

asset allocation 371

asset allocation problem

results, comparing from two formulations 401, 402

asset prices 26

asset returns

stylized facts, investigating 98

Augmented Dickey-Fuller (ADF) 153

auto-ARIMA

best-fitting ARIMA model, finding with 190-202

autocorrelation 102

values, small and decreasing in squared/absolute returns 104-109

AutoETS 173

used, for selecting ETS model 173-175

autoregressive (AR) model 306

AutoRegressive Network (AR-Net) 683

average precision 516

B

backtesting 415
AllInSizer 437, 438
fixed commission per order 439
fixed commission per share 438, 439
backtest, trading strategies
look-ahead bias 415
meeting investment objectives and constraints 416
multiple testing 416
outlier detection and treatment 416
realistic trading environment 416
representative sample period 416
survivorship bias 415
backtrader
backtesting with 431
used, for event-driven backtesting 424-431
backward difference encoder 561
Balanced Random Forest 570
barrier options 361
pricing 361-363
Bayesian hyperparameter optimization 580, 581
libraries 595, 596
running 582-587
shortcomings 581
Bayesian Online Change Point Detection (BOCPD) 91
best-fitting ARIMA model
used, for finding auto-ARIMA 190-203
beta 2, 377
Binary encoder 562
Black-Scholes (BS) 359
bokeh 64
URL 69
Bollinger bands 114, 440
boosting 549
procedures 550

Boruta algorithm 621

Box-Cox transformation 165

Brownian motion 340

built-in cross-validation 259- 261

buy/sell strategy, based on Bollinger bands
backtesting 440- 445

C

Calmar ratio 377

candlestick chart 69

creating 69-72
creating, with mplfinance 72-74
creating, with pure plotly 72- 74

candlestick patterns
recognizing 124-128

Capital Asset Pricing Model (CAPM)
estimating 276, 277
implementing 277- 279

Carhart's four-factor model
estimating 292
implementing 293-, 297
momentum factor 292

Catboost encoder 562

categorical encoders

ML pipelines, fitting with 557-561
library 504, 505

categorical variables
encoding 499-503

CCC-GARCH model

used, for multivariate volatility forecasting 325-329

changepoint 86

detecting, in time series 86-88
detection algorithms, using 90

classification evaluation metrics
exploring 513, 514

class imbalance

approaches, for handling 562- 569

class inheritance 525

CoinGecko

- Bitcoin's current price 23
- data, obtaining from 21, 22
- trending coins 23

combinatorial purged cross-validation algorithm 219

conditional covariance matrix

- forecasting, with DCC-GARCH model 330- 336

conditional heteroskedasticity 305

conditional hyperparameter spaces 588, 589

confusion matrix

- possible values 510

Consumer Price Index (CPI) 30, 158

continuation patterns 124

convex optimization, with cvxpy

- used, for finding efficient frontier 397-400

count encoder 561

cross-entropy loss function 656

cross-sectional factor models

- estimating, with Fama-MacBeth regression 298-303

cross-validation 206

- used, for tuning hyperparameters 529- 536

cumulative distribution function (CDF) 350

currencies

- converting 39, 40

curse of dimensionality 499

D

data, obtaining

- from Alpha Vantage 16- 20
- from CoinGecko 21, 22
- from Intrinio 9, 10, 11
- from Nasdaq Data Link 5-8
- obtaining, from Yahoo Finance 2, 4

data-generating process (DGP) 152

data leakage 488

- handling, with k-fold target encoding 561

dataset

- loading, from CSV file into Python 462-468

DCC-GARCH model

- conditional covariance matrix, forecasting with 330-336

decision trees

- cons 506

- evaluation criterias 511

- fitting 505-513

- pros 506

- visualizing, with dtreeviz 517, 518

- warning 505

DeepAR, Amazon

- used, for time series forecasting 669- 677

DeepVAR model

- training 678-682

discretization 339, 345

downside deviation 377

dtreeviz

- decision trees, visualizing with 517, 518

dummy-variable trap 500

E

encoding categorical features

- alternative approaches, exploring 553-556
- drawbacks 554, 555

ensemble classifiers

- AdaBoost 552

- CatBoost 552

- exploring 544, 545

- Extremely Randomized Trees 552

- Histogram-Based gradient boosting 553

- NGBoost 552

- training 545-549

ensemble models

- averaging methods 545

- boosting methods 545
- entity embedding** 646
- equally-weighted (1/n) portfolio** 372
 - performance, evaluating 373-376
- estimators** 519
- ETS methods**
 - references 175
- European call/put option** 348
- European options**
 - pricing, with Monte Carlo simulations 348- 351
- evaluation or event timestamp** 218
- evening star pattern** 129
- event-driven backtesting** 424
 - with backtrader 424- 431
- excess kurtosis** 109
- exchange-traded fund (ETF)** 376
- Exclusive Feature Bundling (EFB)** 551
- expanders** 137
- explainable AI (XAI)**
 - benefits 623, 624
- explainable AI techniques**
 - exploring 624
 - Individual Conditional Expectation (ICE) 624
 - Partial Dependence Plot (PDP) 624
 - SHapley Additive exPlanations (SHAP) 625
- Exploratory Data Analysis (EDA)** 470
 - carrying out 470- 488
- explored hyperparameters** 590-594
- exponential moving average (EMA)** 72, 114
- exponential smoothing methods**
 - time series, modeling with 166-172
- Extreme Gradient Boosting (XGBoost)**
 - concepts 550
- F**
- factor model**
 - features 275
- Fama-French Factors** 284
- Fama-French's five-factor model**
 - implementing 293-297
 - investment factor 292
 - profitability factor 292
- Fama-French three-factor model**
 - estimating 283- 287
 - market factor (MKT) 283
 - size factor 283
 - value factor 283
- Fama-MacBeth regression** 298
 - reference link 303
 - used, for estimating cross-sectional factor models 298-303
- fastai**
 - features 656, 657
 - reference link 657
 - Tabular Learner, exploring 646-656
- fast Fourier transform (FFT)** 265
- feature engineering**
 - applying, for time series 220- 230
 - examples 220
 - for time series 220
- feature importance**
 - benefits 597, 598
 - disadvantages 598
 - drop column feature importance 598
 - investigating 597
 - permutation feature importance 598
- feature selection**
 - combining, with hyperparameter tuning 621, 622
- feature selection techniques**
 - approaches 620
 - embedded methods 610
 - exploring 610-619
 - filter methods 610
 - wrapper methods 610
- Fisher's kurtosis** 109

Five-factor model

reference link 297

forecast

versus multi-step forecast 695-698

Forex API

reference link 40

FRED database

3 Month T-bill, using from 282

frequency

modifying, of time series data 31- 34

frontier 381

- finding, convex optimization with cvxpy used 397-400
- finding, optimization with spicy used 389-395
- finding, with Monte Carlo simulations 382-387

G**GARCH model 306**

- conditional mean model 315
- conditional volatility model 315, 316
- error distribution 316
- estimation details 336, 337
- multivariate GARCH model 337
- stock returns' volatility, modeling with 312-315
- univariate GARCH model 337
- volatility, forecasting with 316- 324

generative adversarial networks (GANs) 369**geometric Brownian motion (GBM) 341**

- pros 346
- used, for simulating stock price dynamics 340-347

ghost batch normalization 668**ghost feature 80****Gini importance 597****Google**

TabNet, exploring 658

Gradient-based One-Side Sampling (GOSS) 551**Gradient Boosted Trees 549****gradient descent 550****Graphics Processing Units (GPUs) 645****greedy algorithm 505****Greeks**

used, for measuring price sensitivity 352

grid searches

- used, for tuning hyperparameters 529- 536
- with multiple classifiers 539, 540

group time series validation 217**H****Hampel filter 81**

used, for outlier detection 81, 82, 83, 84

Hashing encoder 562**heatmap 380****helmert encoder 561****Heroku**

URL 142

Hierarchical Risk Parity (HRP) 406

- advantages 406
- used, for finding optimal portfolio 406-409

Holt's double exponential smoothing (DOS) 167**Holt's linear trend method 167****Holt-Winters' seasonal smoothing 167****hot-deck imputation 493****Hurst exponent**

- used, for detecting patterns in time series 94-98

hyperparameter

- tuning, with grid searches and cross-validation 529-536

I**iceberg orders 42****identified patterns**

using, as features for model/strategy 128, 129

imbalance bars 48

Individual Conditional Expectation (ICE) 624

advantages 624

disadvantages 624

inflation 28

returns, adjusting for 28-30

information value (IV) 556**interactive visualizations**

creating 64-67

interactive web app

building, for technical analysis with Streamlit 129-138

interpolation methods 37-39

reference link 39

interpretability 624**interquartile range (IQR)** 152, 484**Intrinio**

data, obtaining from 9-11

features 11

isolation forest 571**J****James-Stein encoder** 562**Jensen's alpha** 279**joint hypothesis problem** 279**K****Kaggle**

reference link 541

Kendall's Tau 92**kernel density estimate (KDE)** 379, 473, 590**k-fold cross-validation** 206**k-fold target encoding**

data leakage, handling with 561

KNN

drawbacks 498

kurtosis 377**Kwiatkowski-Phillips-Schmidt-Shin (KPSS)** 153**L****label encoding** 499**Least Squares Monte Carlo (LSMC)**

American options, pricing with 353-356

Leave One Out Encoding (LOOE) 556**leverage effect** 105-109

investigating 109-111

LightGBM

features 551

possibilities 609

linear interpolation 38**line plots**

creating 55-57

Ljung-Box test

reference link 189

Local Interpretable Model-agnostic Explanations (LIME) 641**locally estimated scatterplot smoothing (LOESS)** 149, 151**log returns** 26, 421

calculating 28

long/short strategy based on RSI

backtesting 433-437

M**Mann-Kendall (MK) test** 92**Markov Chain Monte Carlo (MCMC)** 249**Markowitz's curse** 405**Matthew's correlation coefficient** 516**Max drawdown** 376**Maximum Likelihood Estimation (MLE)** 176**Maximum Relevance Minimum Redundancy (MRMR)** 620**maximum Sharpe ratio portfolio** 388, 389**Mean Absolute Error (MAE)** 216

- Mean Absolute Percentage Error (MAPE)** 174, 216
- Mean Decrease in Impurity (MDI)** 597
- mean encoding** 555
- mean-reversion** 94
- Mean Squared Error (MSE)** 216
- mean-variance analysis** 371
- mean-variance portfolio optimization**
backtesting 454-459
- M-estimate encoder** 561
- Meta**
Prophet 248
- metrics**
calculating, to evaluate portfolio's performance 446
- Minimum Variance portfolio** 389
- Minimum Volatility portfolio** 388
- MissForest**
advantages 498
disadvantages 498
- missing at random (MAR)** 492
- missing completely at random (MCAR)** 492
- missingno library**
available visualizations 496, 497
- missing not at random (MNAR)** 492
- missing time series data**
imputing, ways 34- 37
- missing values**
categorizing 492
dealing with 492- 495
identifying 492
- ML-based approaches**
used, for imputing missing values 497, 498
- modern portfolio theory (MPT)** 371
assumptions 372
- momentum factor**
reference link 297
- monotonic constraints** 551
- Monte Carlo**
Value-at-Risk (VaR), estimating with 363- 369
- Monte Carlo simulations**
used, for finding efficient frontier 382-387
used, for improving valuation function 351, 352
used, for pricing European options 348-351
- moving average convergence divergence (MACD)** 114
downloading, with API 122, 124
- moving average crossover strategy**
backtesting, with crypto data 447- 453
- multiple imputation approaches** 497
- Multiple Imputation by Chained Equations (MICE)** 497
- Multiple Seasonal-Trend Decomposition using LOESS (MSTL)** 149
- multiplicative model** 145
- multivariate GARCH model**
estimation, parallelizing 337, 338
- multivariate volatility forecasting**
with CCC-GARCH model 325-329
- Mutual Information (MI) score** 618
- ## N
- Nasdaq Data Link**
data, obtaining from 5-8
reference link 6
- nearest neighbors imputation** 497
- negative skewness (third moment)** 99
- nested cross-validation** 208
- NeuralProphet**
features 698
forecast, versus multi-step forecast 695- 698
holidays and special events, adding 694, 695
time series forecasting 682- 693
- non-Gaussian distribution of returns** 99, 100, 101, 108

- descriptive statistics 109
histogram, of returns 108
Q-Q plot 108
- non-stationary data**
drawbacks 153
- non-systematic component**
noise 144
- O**
- Omega ratio** 376
- OneHotEncoder**
categories, specifying for 504
- one-hot encoding** 499
issues 554
pandas, using 503
warning 505
- Open, High, Low, and Close (OHLC)** 4
prices 70
- optimal portfolio**
finding, with Hierarchical Risk Parity 406- 409
- optimization, with scipy**
used, for finding efficient frontier 389-395
- oracle approximating shrinkage (OAS)** 409
- ordinal encoding** 561
- ordinary least squares (OLS)** 176
- Ornstein-Uhlenbeck process** 94
- outlier detection**
with Hampel filter 81-84
with rolling statistics 77-80
- outliers** 77
identifying, with stock returns 84-86
- oversampling methods**
Borderline SMOTE 570
K-means SMOTE 571
SVM SMOTE 571
Synthetic Minority Oversampling Technique for Nominal and Continuous (SMOTE-NC) 570
- P**
- pandas**
using, for one-hot encoding 504
vectorized backtesting with 417-421
- Partial Dependence Plot (PDP)**
advantages 625
disadvantages 625
- patterns**
detecting, in time series with Hurst exponent 94-98
- Pearson's correlation coefficient** 484
- permutation feature importance** 598
pros and cons 599
- Phillips-Perron (PP)** 157
- pipelines**
benefits 519
building 520-523
custom transformers, adding 524-528
elements, accessing 528
used, for organizing projects 519, 520
- Platform as a Service (PaaS)** 142
- point anomaly detection** 78
- portfolio rebalancing** 291
- Precision-Recall curve** 513
analyzing 514-517
- prices**
converting, to returns 25-27
- Principal Components Analysis (PCA)** 565
- probability density function (PDF)** 100, 352
- Proof of Concept (PoC)** 504
- Prophet** 249
features 248
model, tuning 261
used, in forecasting 248- 258
- PyCaret**
features 272
URL 86

using, for time series forecasting 262-272
pycoingecko library
reference link 23
pyfolio 447
PyPortfolioOpt 410
efficient frontier, obtaining 410-412
Python libraries, on AI explainability
investigating 642
PyTorch Forecasting 677

Q

quantile plot 380
quantile-quantile (Q-Q) plot 100, 108
QuantLib
American options, pricing with 357-361
quantstats 377
pandas DataFrames/Series, enriching with new methods 380
quarter plot 62

R

Random Forest model
feature importance, evaluating 599-608
random oversampling 563
random search (randomized grid search) 530
random undersampling 563
random walk 94
realized volatility 32
Receiver Operating Characteristic (ROC) 511
Rectified Linear Unit (ReLU) 655
Recursive Feature Elimination (RFE) 619
reduced regression
time series, forecasting as 235-247
reduction process 235
relative strength index (RSI) 114, 433
rescaled range (R/S) analysis 98

returns
adjusting, for inflation 28-30
benefit 26
log returns 26
prices, converting to 25-28
simple returns 26
reversal patterns 124
RobustStatDetector 90
Rolling Sharpe ratio 378
rolling statistics
used, for outlier detection 77-80
rolling three-factor model
implementing 288-291
Root Mean Squared Error (RMSE) 216

S

seaborn 61
seasonal decomposition
approaches 152
seasonality 58
seasonal patterns
additional information, visualizing 61-63
visualizing 58-60
sequential attention 658
Sequential Least-Squares Programming (SLSQP) algorithm 394
Sequential Model-Based Optimization (SMBO) 580
serial correlation 102
SHapley Additive exPlanations (SHAP) 625, 626
advantages 626
disadvantages 626
Sharpe ratio 376
signal types 436
simple exponential smoothing (SES) 167
Simple Moving Average (SMA) 72, 115, 417
calculating 431, 432

- simple returns** 26
 - calculating 27
 - Simplified Wrapper and Interface Generator (SWIG)** 357
 - single imputation approaches** 497
 - Singular Value Decomposition (SVD)** 368
 - sizers**
 - reference link 440
 - skew** 377
 - sktime**
 - advantages 247
 - documentation link 86
 - slippage** 416
 - Sortino ratio** 376
 - sparsemax** 668
 - squared/absolute returns**
 - autocorrelation values, small and decreasing 104-109
 - stacked ensemble**
 - creating 573-579
 - stacking** 573, 574
 - goal 573
 - stationarity** 25
 - correcting, in time series 158-164
 - testing, in time series 152-158
 - STL decomposition** 150, 151
 - advantages 149
 - stochastic differential equations (SDEs)** 340
 - Stochastic Gradient Boosted Trees** 550
 - stock-keeping units (SKUs)** 669
 - stock price dynamics**
 - simulating, with GBM 340-347
 - stock return's volatility**
 - modeling, with ARCH models 306-311
 - modeling, with GARCH models 312-315
 - Streamlit**
 - documentation link 139
 - reference link** 139
 - sign up page, reference link** 139
 - Streamlit cloud, reference link** 142
 - using, to build interactive web app for technical analysis** 129-138
 - stylized facts** 98
 - absence, of autocorrelation in returns 102-109
 - autocorrelation, small and decreasing in squared/absolute returns 104-109
 - investigating, of asset returns 98
 - leverage effect 105-109
 - non-Gaussian distribution of returns 99-108
 - volatility clustering 102, 109
 - successive halving**
 - used, for performing faster search 536-538
 - sum encoder** 561
 - surrogate model** 580
 - symmetric MAPE (sMAPE)** 245
 - Synthetic Minority Oversampling Technique for Nominal and Continuous (SMOTE-NC)** 570
 - Synthetic Minority Oversampling Technique (SMOTE)** 563
 - systematic components**
 - level 144
 - seasonality 144
 - trend 144
- ## T
- TabNet, Google**
 - exploring 658- 668
 - features 658, 659
 - implementation, in PyTorch 668
 - Tabular Learner, fastai**
 - exploring 646-656
 - tail ratio** 377
 - TA-Lib** 114, 119
 - reference link 129
 - URL 120

- tangency portfolio** 387
- target encoding** 555
- target orders** 453
- technical analysis (TA)** 113
- deploying 139-142
 - interactive web app, building with Streamlit 129-138
- technical indicators**
- calculating 113-119
 - downloading 120-122
- techniques, for tackling overfitting**
- batch normalization 655
 - dropout 655
 - weight decay 655
- term frequency-inverse document frequency (TF-IDF)** 523
- test sets**
- data, splitting into 488-492
- Three-Factor Model** 284
- tick bars** 43
- time bars**
- drawbacks 42
- time-related features**
- creating 230-235
- time series**
- changepoints, detecting 86-88
 - feature engineering, applying 220-230
 - forecasting as reduced regression 235-247
 - modeling, with ARIMA class models 176-189
 - modeling, with exponential smoothing methods 166-173
 - non-systematic components 144
 - patterns, detecting with Hurst exponent 94-98
 - stationarity, correcting 158-164
 - stationarity, testing for 152-158
 - systematic components 144
 - trends, detecting 92-94
 - validation methods 206-216
- time series data**
- frequency, modifying of 31- 34
 - visualizing 52- 54
- time series decomposition**
- goals 144
 - performing 146-148
 - references 152
- time series forecast accuracy**
- evaluating, metrics 216
- time series forecasting**
- NeuralProphet, using for 683-693
 - with Amazon's DeepAR 669-677
 - with PyCaret 262-272
- trade data**
- aggregating, ways 43-48
- training sets**
- data, splitting into 488-492
- transformations, applying to data**
- continuous variables, discretizing 524
 - numerical features, scaling 524
 - outliers, transforming/removing 524
- transformers** 519
- treeinterpreter** 642
- tree-structured Parzen Estimator (TPE)** 580
- trends**
- detecting, in time series 92-94
- Tukey's fences** 484
- U**
- undersampling methods**
- Edited Nearest Neighbors 570
 - NearMiss 570
 - Tomek links 570
- underwater plot** 376

V**validation methods**

for time series 206- 216

validation set 490**valuation function**

improving, with Monte Carlo simulations 351, 352

Value-at-Risk (VaR) 305

estimating, with Monte Carlo 363-369

Variance Inflation Factor (VIF) 620**variance targeting** 336**variance thresholding** 620**vectorized backtesting** 417

transaction costs, accounting 422, 423

with pandas 417- 421

volatility clustering 102, 109**volatility forecasting**

analytical approach 317

bootstrap forecasts 317

GARCH models, using 316-324

simulation-based forecasts 317

Volatility Index (VIX) 305**volatility trading** 305**volume bars** 43**volume-weighted average price (VWAP)** 48**W****walk-forward validation** 207

used, for calculating model performance 208

weak stationarity 153**Weight of Evidence (WoE) encoding** 556**winsorization** 81**wrapper techniques**

backward feature selection 620

considerations 621

exhaustive feature selection 621

forward feature selection 620
stepwise selection 621

X**XGBoost**

possibilities 609

predictions, explaining 627-641

Y**Yahoo Finance**

data, obtaining from 2, 4

libraries 5

reference link 5

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781803243191>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

