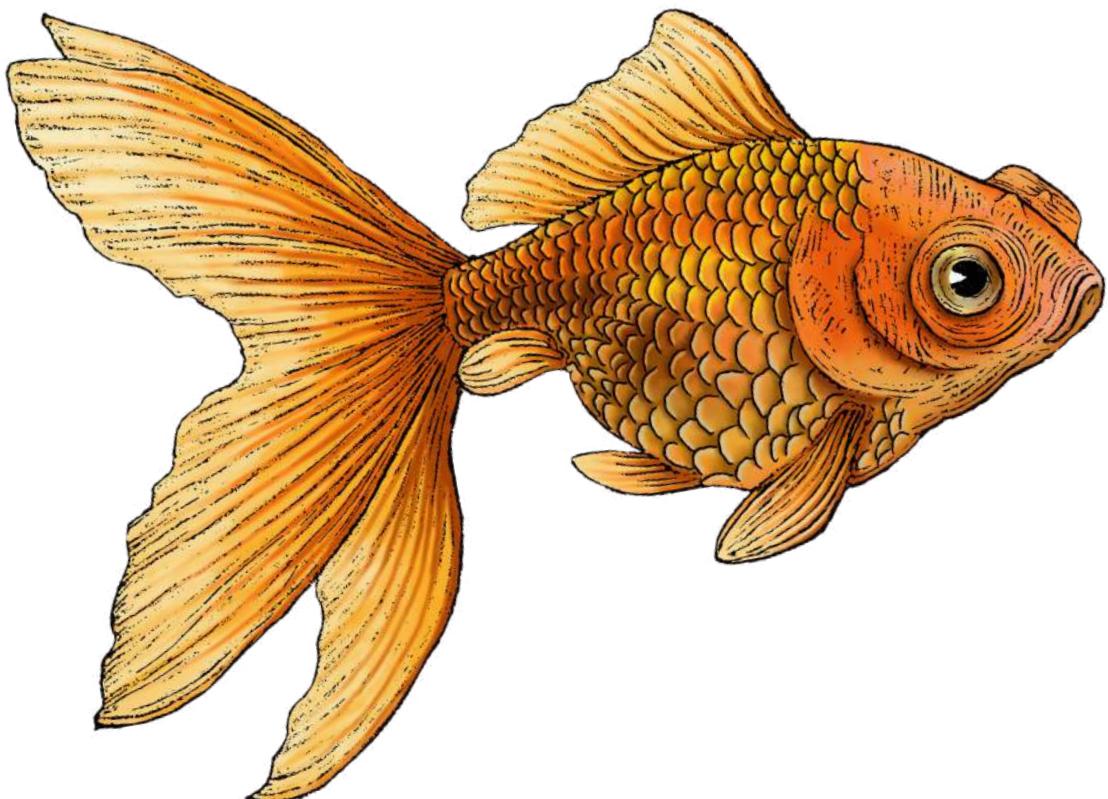


O'REILLY®

Deep Learning for Finance

Creating Machine & Deep Learning Models
for Trading in Python



Sofien Kaabar

Deep Learning for Finance

Deep learning is rapidly gaining momentum in the world of finance and trading. But for many professional traders, this sophisticated field has a reputation for being complex and difficult. This hands-on guide teaches you how to develop a deep learning trading model from scratch using Python, and it also helps you create and backtest trading algorithms based on machine learning and reinforcement learning.

Sofien Kaabar—financial author, trading consultant, and institutional market strategist—introduces deep learning strategies that combine technical and quantitative analyses. By fusing deep learning concepts with technical analysis, this unique book presents outside-the-box ideas in the world of financial trading. This A-Z guide also includes a full introduction to technical analysis, evaluating machine learning algorithms, and algorithm optimization.

- Understand and create machine learning and deep learning models
- Explore the details behind reinforcement learning and see how it's used in time series
- Understand how to interpret performance evaluation metrics
- Examine technical analysis and learn how it works in financial markets
- Create technical indicators in Python and combine them with ML models for optimization
- Evaluate the models' profitability and predictability to understand their limitations and potential

"This book is a magisterial work that stands as a landmark in the field of quantitative trading, data science, and financial algorithms."

—Amaury Goguel
Head of MSc Financial Markets & Investments,
SKEMA Business School, Paris

"This is the book I wish I had read when I started developing ML trading algorithms as a quantitative investment strategist."

—Ning Wang
Quantitative Investment Structurer,
Barclays

Sofien Kaabar is a financial author, trading consultant, and institutional market strategist specializing in the currencies market with a focus on technical and quantitative topics. Sofien's goal is to make technical analysis objective by incorporating clear conditions that can be analyzed and created with the use of technical indicators.

DATA

US \$69.99 CAN \$87.99

ISBN: 978-1-098-14839-3

Twitter: @oreillymedia
[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)



9 781098148393

Praise for *Deep Learning for Finance*

As the scientific director of a leading program in market finance for over 10 years, I can testify to the immense quality of this book. *Deep Learning for Finance* is a magisterial work that stands as a landmark in the field of quantitative trading, data science, and financial algorithms. The author's profound knowledge and deep insights are evident throughout the book, which is written with clarity and precision. It will undoubtedly become a reference in this specialized field in which the inherent complexity of the subject is rarely well served in disclosure books. This one is an exception, striking the perfect balance between clarity and precision without becoming oversimplistic or overcomplex. It is an essential read for anyone interested in the cutting edge of quantitative trading/finance, both for master's degree students in finance and for practitioners.

—Amaury Goguel, Head of MSc Financial Markets & Investments, SKEMA Business School, Paris

This is the book I wish I had read when I started developing ML trading algorithms as a quantitative investment strategist.

—Ning Wang, Quantitative Investment Structurer, Barclays

Sofien is a master, providing the right balance of detail and autonomy, allowing readers to connect the dots themselves.

—Timothy Kipper, Head of Business Development, Coperniq.io

Deep Learning for Finance

*Creating Machine and Deep Learning Models
for Trading in Python*

Sofien Kaabar

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Deep Learning for Finance

by Sofien Kaabar

Copyright © 2024 Sofien Kaabar. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Michelle Smith

Indexer: WordCo Indexing Services, Inc.

Development Editor: Corbin Collins

Interior Designer: David Futato

Production Editor: Elizabeth Faeirm

Cover Designer: Karen Montgomery

Copyeditor: Audrey Doyle

Illustrator: Kate Dullea

Proofreader: Piper Editorial Consulting, LLC

January 2024: First Edition

Revision History for the First Edition

2024-01-08: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098148393> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Deep Learning for Finance*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14839-3

[LSI]

Table of Contents

Preface.....	ix
1. Introducing Data Science and Trading.....	1
Understanding Data	1
Understanding Data Science	10
Introduction to Financial Markets and Trading	14
Applications of Data Science in Finance	17
Summary	18
2. Essential Probabilistic Methods for Deep Learning.....	19
A Primer on Probability	19
Introduction to Probabilistic Concepts	20
Sampling and Hypothesis Testing	25
A Primer on Information Theory	30
Summary	34
3. Descriptive Statistics and Data Analysis.....	35
Measures of Central Tendency	36
Measures of Variability	41
Measures of Shape	45
Visualizing Data	54
Correlation	63
The Concept of Stationarity	70
Regression Analysis and Statistical Inference	77
Summary	80

4. Linear Algebra and Calculus for Deep Learning.....	81
Linear Algebra	82
Vectors and Matrices	82
Introduction to Linear Equations	92
Systems of Equations	96
Trigonometry	101
Calculus	105
Limits and Continuity	105
Derivatives	114
Integrals and the Fundamental Theorem of Calculus	124
Optimization	128
Summary	133
5. Introducing Technical Analysis.....	135
Charting Analysis	137
Indicator Analysis	143
Moving Averages	143
The Relative Strength Index	145
Pattern Recognition	147
Summary	149
6. Introductory Python for Data Science.....	151
Downloading Python	151
Basic Operations and Syntax	153
Control Flow	157
Libraries and Functions	159
Exception Handling and Errors	163
Data Structures in numpy and pandas	166
Importing Financial Time Series in Python	170
Summary	175
7. Machine Learning Models for Time Series Prediction.....	177
The Framework	177
Machine Learning Models	190
Linear Regression	190
Support Vector Regression	194
Stochastic Gradient Descent Regression	197
Nearest Neighbors Regression	200
Decision Tree Regression	203
Random Forest Regression	205

AdaBoost Regression	207
XGBoost Regression	209
Overfitting and Underfitting	211
Summary	214
8. Deep Learning for Time Series Prediction I.....	215
A Walk Through Neural Networks	215
Activation Functions	217
Backpropagation	224
Optimization Algorithms	226
Regularization Techniques	226
Multilayer Perceptrons	227
Recurrent Neural Networks	232
Long Short-Term Memory	235
Temporal Convolutional Neural Networks	243
Summary	246
9. Deep Learning for Time Series Prediction II.....	249
Fractional Differentiation	249
Forecasting Threshold	254
Continuous Retraining	256
Time Series Cross Validation	259
Multiperiod Forecasting	261
Applying Regularization to MLPs	271
Summary	276
10. Deep Reinforcement Learning for Time Series Prediction.....	277
Intuition of Reinforcement Learning	278
Deep Reinforcement Learning	284
Summary	290
11. Advanced Techniques and Strategies.....	291
Using COT Data to Predict Long-Term Trends	291
Algorithm 1: Indirect One-Step COT Model	297
Algorithm 2: MPF COT Direct Model	299
Algorithm 3: MPF COT Recursive Model	301
Putting It All Together	303
Using Technical Indicators as Inputs	304
Predicting Bitcoin's Volatility Using Deep Learning	308
Real-Time Visualization of Training	317
Summary	324

12. Market Drivers and Risk Management.....	325
Market Drivers	325
Market Drivers and Economic Intuition	326
News Interpretation	328
Risk Management	330
Basics of Risk Management	331
Behavioral Finance: The Power of Biases	333
Summary	336
Index.....	337

Preface

Learning never exhausts the mind.

—Leonardo da Vinci

Machine learning and deep learning have completely changed the finance industry in recent years. The different learning models are well suited to a world where data is abundant and continuous. Data is the new gold, and its value keeps rising as proper analyses lead to key business decisions, which are the driver of economic shifts.

The rise of quantitative funds is living proof that the world of data science has much to offer to the trading world. After fundamental and technical traders, a new breed of leaders of the universe is emerging. These are the quantitative traders who rely on machine-based algorithms with extremely complex operations that seek to forecast and outperform the markets.

This book covers in detail the subject of *deep learning for finance*.

Why This Book?

I have spent my career researching trading strategies, techniques, and all things related to the financial world. Through the years, I have become familiar with a few algorithmic models that have the potential of adding value to the trading framework. In this book, I discuss different learning models and their applications in the trading world, with a focus on deep learning and neural networks. My main aim is to cover them in such a way that everyone understands how they function.

Machines can perform operations and detection better than humans for many reasons, one of which is their objectivity. This means that one of the key skills you will learn is how to use Python to create the algorithms required to do such operations.

As mentioned, my objective is to provide a comprehensive introduction to the use of deep learning in finance. I do this by discussing a wide range of topics, including data science, trading, machine and deep learning models, and reinforcement learning applications for trading.

The book begins with an overview of the field of data science and its role in the finance world. It then delves into the knowledge requirements, such as statistics, math, and Python, before focusing on how to use machine and deep learning in trading strategies.

Who Should Read It?

This book is intended for a wide audience, including professionals and academics in finance, data scientists, quantitative traders, and students of finance of any level. It provides a thorough introduction to the use of machine and deep learning in time series prediction, and it is an essential resource for anyone who wants to understand and apply these powerful techniques.

The book assumes you have basic background knowledge in both Python programming (professional Python users will find the code very straightforward) and financial trading. I take a clear and simple approach that focuses on the key concepts so that you understand the purpose of every idea.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at <https://github.com/sofienkaabar/deep-learning-for-finance>.

If you have a technical question or a problem using the code examples, please send email to bookquestions@oreilly.com.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution, which usually includes the title, author, publisher, and ISBN. For example: “*Deep Learning for Finance* by Sofien Kaabar (O'Reilly). Copyright 2024 Sofien Kaabar, 978-1-098-14839-3.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning



For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
<https://www.oreilly.com/about/contact.html>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/deep-learning-for-finance>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>.

Follow us on Twitter: <https://twitter.com/oreillymedia>.

Watch us on YouTube: <https://youtube.com/oreillymedia>.

Acknowledgments

Nothing would be the same without the support of my parents, which is why I can't help but acknowledge their direct and indirect impact on the book.

I would also like to acknowledge the debt I owe to the editors, Michelle Smith and Corbin Collins, as well as to the production editor, Elizabeth Faerm, for their continued support, the amazing job they do, and their patience. Similarly, I would like to thank every person at O'Reilly who was involved in the production of this book.

Additionally, my special thanks go to the great technical reviewers for their immense contributions. They had a sizable impact on making this book readable, useful, and straightforward. I could not ask for better people to review my book.

Finally, I am deeply grateful to you, the reader, for investing your time into reading my work and for placing your trust in my research. I hope you find it useful.

Introducing Data Science and Trading

The best way to begin learning about a complex topic is to break it down into smaller parts and understand those pieces first. Understanding deep learning for finance requires knowledge of data science and financial markets.

This chapter lays the building blocks needed to thoroughly understand data science and its uses, as well as to understand financial markets and how trading and forecasting can benefit from data science.

By the end of the chapter, you should know what data science is, what its applications are, and how you can use it in finance to extract value.

Understanding Data

It is impossible to understand the field of data science without first understanding the types and structures of data. After all, the first word for the name of this immense field is *data*. So, what is data? And more importantly, what can you do with it?

Data in its simplest and purest form is a collection of raw information that can be of any type (numerical, text, boolean, etc.).

The final aim of collecting data is decision-making. This is done through a complex process that ranges from the act of gathering and processing data to interpreting it and using the results to make a decision.

Let's take an example of using data to make a decision. Suppose you have a portfolio composed of five different equally weighted dividend-paying stocks, as detailed in [Table 1-1](#).

Table 1-1. Stocks and their dividend yields

Stock	Dividend yield
A	5.20%
B	3.99%
C	4.12%
D	6.94%
E	5.55%



A *dividend* is the payment made to shareholders from a company's profits. The *dividend yield* is the amount distributed in monetary units over the current share price of the company.

Analyzing this data can help you understand the average dividend yield you are receiving from your portfolio. The average is basically the sum divided by the quantity, and it gives a quick snapshot of the overall dividend yield of the portfolio:

$$\text{Average dividend yield} = \frac{5.20\% + 3.99\% + 4.12\% + 6.94\% + 5.55\%}{5} = 5.16\%$$

Therefore, the average dividend yield of your portfolio is 5.16%. This information can help you compare your average dividend yield to other portfolios so that you know whether you have to make any adjustments.

Another metric you can calculate is the number of stocks held in the portfolio. This may provide the first informational brick in constructing a wall of diversification. Even though these two pieces of information (average dividend yield and the number of stocks in the portfolio) are very simple, complex data analysis begins with simple metrics and may sometimes not require sophisticated models to properly interpret the situation.

The two metrics you calculated in the previous example are called the *average* (or mean) and the *count* (or number of elements). They are part of a field called *descriptive statistics* discussed in [Chapter 3](#), which is also itself part of data science.

Let's take another example of data analysis for inferential purposes. Suppose you have calculated a yearly correlation measure between two commodities, and you want to predict whether the next yearly correlation will be positive or negative. [Table 1-2](#) has the details of the calculations.

Table 1-2. Yearly correlation measures

Year	Correlation
2015	Positive
2016	Positive
2017	Positive
2018	Negative
2019	Positive
2020	Positive
2021	Positive
2022	Positive
2023	Positive



Correlation is a measure of the linear reliance between two time series. A *positive correlation* generally means that the two time series move on average in the same direction, while a *negative correlation* generally means that the two time series move on average in opposite directions. Correlation is discussed in [Chapter 3](#).

From [Table 1-2](#), the historical correlation between the two commodities was mostly (i.e., 88%) positive. Taking into account historical observations, you can say that there is an 88% probability that the next correlation measure will be positive. This also means that there is a 12% probability that the next correlation measure will be negative:

$$E(\text{Positive correlation}) = \frac{8}{9} = 88.88\%$$

This is another basic example of how to use data draw inferences from observations and make decisions. Of course, the assumption here is that historical results will exactly reflect future results, which is unlikely in real life, but occasionally, to predict the future all you have is the past.

Now, before discussing data science, let's review what types of data can be used and segment them into different groups:

Numerical data

This type of data is composed of numbers that reflect a certain type of information that is collected at regular or irregular intervals. Examples can include market data (OHLC,¹ volume, spreads, etc.) and financial statements data (assets, revenue, costs, etc.).

¹ OHLC refers to the four essential pieces of market data: open price, high price, low price, and close price.

Categorical data

Categorical data is data that can be organized into groups or categories using names or labels. It is qualitative rather than quantitative. For example, the blood type of patients is a type of categorical data. Another example is the eye color of different samples from a population.

Text data

Text data has been on the rise in recent years with the development of *natural language processing* (NLP). Machine learning models use text data to translate, interpret, and analyze the sentiment of the text.

Visual data

Images and videos are also considered data, and you can process and transform them into valuable information. For example, a *convolutional neural network* (CNN) is a type of algorithm (discussed in [Chapter 8](#)) that can recognize and categorize photos by labels (e.g., labeling cat photos as cats).

Audio data

Audio data is very valuable and can help save time on transcriptions. For example, you can use algorithms on audio to create captions and automatically create subtitles. You can also create models that interpret the sentiment of the speaker using the tone and volume of the audio.

Data science is a transdisciplinary field that tries to extract intelligence and conclusions from data using different techniques and models, be they simple or complex. The data science process is composed of many steps besides just analyzing data. The following summarizes these steps:

1. *Data gathering*: This process involves the acquisition of data from reliable and accurate sources. A widely known phrase in computer science generally credited to George Fuechsel goes “Garbage in, garbage out,” and it refers to the need to have quality data that you can rely on for proper analysis. Basically, if you have inaccurate or faulty data, then all your processes will be invalid.
2. *Data preprocessing*: Occasionally, the data you acquire can be in a raw form, and it needs to be preprocessed and cleaned for the data science models to be able to use it. For example, dropping unnecessary data, adding missing values, or eliminating invalid and duplicate data can be part of the preprocessing step. Other, more complex examples can include *normalization* and *denoising* of data. The aim of this step is to get the data ready for analysis.
3. *Data exploration*: During this step, basic statistical research is conducted to find trends and other characteristics in data. An example of data exploration is to calculate the mean of the data.

4. *Data visualization*: This is an important step that is an add-on to the previous step. It includes creating visualizations such as histograms and heatmaps to help identify patterns and trends and facilitate interpretation.
5. *Data analysis*: This is the main focus of the data science process. This is when you *fit* (train) the data using different learning models so that they interpret and predict the future outcome based on the given parameters.
6. *Data interpretation*: This step deals with understanding the feedback and conclusions presented by the data science models. *Optimization* may also be a part of this step; in those cases, we loop back to step 5 and run the models again with the updated parameters before reinterpreting them and evaluating the performance.

Let's take a simple example in Python that applies the steps of the data science process. Suppose you want to analyze and predict the VIX (*volatility index*), a volatility time series indicator that represents the implied volatility of the S&P 500 stock market index. The VIX has been available since 1993 and is issued by the Chicago Board Options Exchange (CBOE).



There is also a hidden step in the data science process that I refer to as *step zero*, and it occurs when you form an idea based on which process should be initiated. After all, you wouldn't be applying the process if you didn't have a motive first. For example, believing that inflation numbers may drive the returns of certain commodities is an idea and a motive to start exploring the data in search of real numbers that prove this hypothesis.

Because it is meant to measure the level of fear or uncertainty in the stock market, the VIX is frequently referred to as the *fear index*. It is a percentage that is computed using the pricing of options on the S&P 500. A higher VIX value correlates with greater market turbulence and uncertainty, whereas a lower value correlates with greater stability on average.

The first step is data gathering, which in this case can be automated using Python. The following code block connects to the website of the Federal Reserve of Saint Louis and downloads the historical data of the VIX between January 1, 1990, and January 23, 2023 ([Chapter 6](#) is dedicated to introducing Python and writing code; for the moment, you do not have to understand the code, as that is not yet the goal):

```
# Importing the required library
import pandas_datareader as pdr
# Setting the beginning and end of the historical data
start_date = '1990-01-01'
end_date   = '2023-01-23'
# Creating a dataframe and downloading the VIX data
vix = pdr.DataReader('VIXCLS', 'fred', start_date, end_date)
```

```
# Printing the latest five observations of the dataframe
print(vix.tail())
```

The code uses the *pandas* library to import the DataReader function, which fetches the historical data online from a variety of sources. The DataReader function takes the name of the data as the first argument, followed by the source and the dates. The output of `print(vix.tail())` is shown in [Table 1-3](#).

Table 1-3. Output of print(vix.tail())

DATE	VIXCLS
2023-01-17	19.36
2023-01-01	20.34
2023-01-19	20.52
2023-01-20	19.85
2023-01-23	19.81

Let's move on to the second step: data preprocessing. I divide this part into checking for invalid data and transforming the data so that it is ready for use. When dealing with time series, especially downloaded time series, you may sometimes encounter `nan` values. `NaN` stands for *Not a Number*, and `nan` values occur due to missing, invalid, or corrupt data.

You can deal with `nan` values in many ways. For the sake of this example, let's use the simplest way of dealing with these invalid values, which is to eliminate them. But first, let's write some simple code that outputs the number of `nan` values in the dataframe so that you have an idea of how many values you will delete:

```
# Calculating the number of nan values
count_nan = vix['VIXCLS'].isnull().sum()
# Printing the result
print('Number of nan values in the VIX dataframe: ' + str(count_nan))
```

The code uses the `isnull()` function and sums the number it gets, which gives the number of `nan` values. The output of the previous code snippet is as follows:

```
Number of nan values in the VIX dataframe: 292
```

Now that you have an idea of how many rows you will delete, you can use the following code to drop the invalid rows:

```
# Dropping the nan values from the rows
vix = vix.dropna()
```

The second part of the second step is to transform the data. Data science models typically require *stationary* data, which is data with stable statistical properties such as the mean.



The concept of *stationarity* and the required statistics metrics are discussed in detail in [Chapter 3](#). For now, all you need to know is that it is likely that you will have to transform your raw data into stationary data when using data science models.

To transform the VIX data into stationary data, you can simply take the differences from one value relative to the previous value. The following code snippet takes the VIX dataframe and transforms it into theoretically implied stationary data.²

```
# Taking the differences in an attempt to make the data stationary
vix = vix.diff(periods = 1, axis = 0)
# Dropping the first value of the dataframe
vix = vix.iloc[1: , :]
```

The third step is data exploration, which is all about understanding the data you have in front of you, statistically speaking. As you will see statistical metrics in detail in [Chapter 3](#), I'll limit the discussion to just calculating the mean of the dataset.

The *mean* is simply the value that can represent the other values in the dataset if they were to elect a leader. It is the sum of the values divided by their quantity. The mean is the simplest stat in the descriptive statistics world, and it is definitely the most used one. The following formula shows the mathematical representation of the mean of a set of values:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

You can easily calculate the mean of the dataset as follows:

```
# Calculating the mean of the dataset
mean = vix["VIXCLS"].mean()
# Printing the result
print('The mean of the dataset = ' + str(mean))
```

The output of the previous code snippet is as follows:

```
The mean of the dataset = 0.0003
```

The next step is data visualization, which is mostly considered to be the fun step. Let's chart the VIX's differenced values through time. The following code snippet plots the VIX data shown in [Figure 1-1](#):

```
# Importing the required library
import matplotlib.pyplot as plt
# Plotting the latest 250 observations in black with a label
```

² The reason I am saying “implied” is that stationarity must be verified through statistical checks that you will see in [Chapter 3](#). At the moment, the assumption is that differencing the data gives stationary time series.

```

plt.plot(vix[-250:], color = 'black', linewidth = 1.5,
         label = 'Change in VIX')
# Plotting a red dashed horizontal line that is equal to mean
plt.axhline(y = mean, color = 'red', linestyle = 'dashed')
# Calling a grid to facilitate the visual component
plt.grid()
# Calling the legend function so it appears with the chart
plt.legend()
# Calling the plot
plt.show()

```

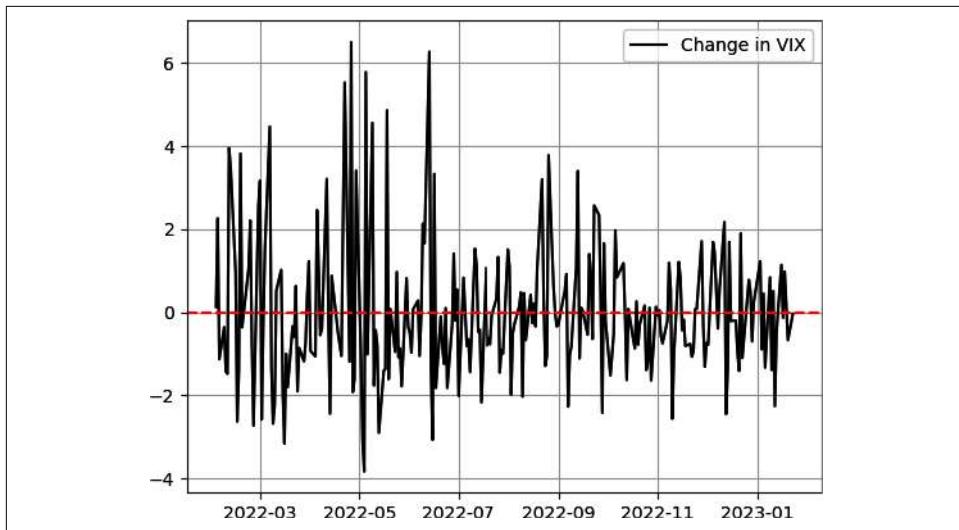


Figure 1-1. Change in the VIX since early 2022

Steps 5 and 6, data analysis and data interpretation, are what you are going to study thoroughly in this book, so let's skip them for now and concentrate on the introductory part of data science.

Let's go back to the invalid or missing data problem before moving on. Sometimes data is incomplete and has missing cells. Even though this has the potential to hinder the predictive ability of the algorithm, it should not stop you from continuing the analysis as there are quick fixes that help lessen the negative impact of the empty cells. For instance, consider **Table 1-4**.

Table 1-4. Quarterly GDP

Quarter	GDP
Q1 2020	0.9%
Q2 2020	1.2%
Q3 2020	0.5%
Q4 2020	0.4%

Quarter	GDP
Q1 2021	#N/A
Q2 2021	1.0%
Q3 2021	1.1%
Q4 2021	0.6%

The table contains the quarterly gross domestic product (GDP)³ of a hypothetical country. Notice how the table is missing the value for Q1 2021. There are three basic ways to solve this issue:

Delete the cell that contains the missing value.

This is the technique used in the VIX example. It simply considers that the timestamp does not exist. It is the easiest fix.

Assume that the missing cell is equal to the previous cell.

This is also a simple fix that has the aim of smoothing the data instead of completely ignoring the issue.

Calculate a mean or a median of the cells around the empty value.

This technique takes smoothing one step further and assumes that the missing value is equal to the mean between the previous and next values. Additionally, it can be the mean of a few past observations.

Data science comprises a range of mathematical and statistical concepts and requires a deep understanding of machine learning algorithms. In this book, these concepts are discussed in detail but also in an easy-to-grasp manner to benefit both technical and nontechnical readers. Many models are assumed to be mystery boxes, and there is a hint of truth in this, but the job of a data scientist is to understand the models before interpreting their results. This helps in understanding the limitations of the models.

This book uses Python as the go-to programming language to create the algorithms. As mentioned, [Chapter 6](#) introduces Python and the knowledge required to manipulate and analyze the data, but it also provides the foundations for creating the different models, which, as you will see, are simpler than you might expect.

Before moving on to the next section, let's have a look at the concept of data storage. After all, data is valuable, but you need to store it in a place where it can be easily fetched and analyzed.

³ The GDP measure is discussed in more detail in [Chapter 12](#).

Data storage refers to the techniques and areas used to store and organize data for future analysis. Data is stored in many formats, such as CSV and XLSX. Other types of formats may include XML, JSON, and even JPEG for images. The format is chosen according to the structure and organization of the data.

Data can also be stored in the cloud or on premises, depending on your storage capacity and costs. For example, you may want to keep your historical, one minute's worth of Apple stock data in the cloud, instead of in a CSV file, so that you save space on your local computer.

When dealing with time series in Python, you are mostly going to deal with two types of data storage: arrays and dataframes. Let's take a look at what they are:

Array

An *array* is used to store elements of the same kind. Typically, a homogeneous dataset (such as numbers) is best kept in an array.

Dataframe

A *dataframe* is a two-dimensional structure that can hold data of various types. It can be compared to a table with columns and rows.

In general, arrays should be used whenever a homogeneous data collection needs to be efficiently stored. When dealing with heterogeneous data or when you need to edit and analyze data in a tabular manner, you should use dataframes.



Data science is continually evolving. New storage methods are being developed all the time in an attempt to make them more efficient and increase their capacity and speed.

Understanding Data Science

Data science plays an essential role in technology and progress. Algorithms rely on information provided from data science tools to perform their tasks. But what are algorithms?

An *algorithm* is a set of ordered procedures that are designed to complete a certain activity or address a particular issue. An algorithm can be as simple as a coin flip or as sophisticated as the Risch algorithm.⁴

Let's take a very simple algorithm that updates a charting platform with the necessary financial data. This algorithm would follow these steps:

⁴ The Risch algorithm is an indefinite integration technique used to find antiderivatives, a concept you will see in [Chapter 4](#).

1. Connect the server and the online data provider.
2. Copy the financial data with the most recent timestamp.
3. Paste the data into the charting platform.
4. Loop back to step 1 and redo the whole process.

That is the nature of algorithms: performing a certain set of instructions with a finite or an infinite goal.



The six data science steps discussed in the previous section can also be considered an algorithm.

Trading strategies are also algorithms, as they have clear rules for the initiation and liquidation of positions. An example of a trading strategy is market arbitrage.

Arbitrage is a type of trading strategy that aims to profit from price differences of the same asset quoted on different exchanges. These price differences are anomalies that are erased by arbitrageurs through their buying and selling activities. Consider a stock that is traded on exchange A and exchange B in different countries (for simplicity reasons, the two countries use the same currency). Naturally, the stock must trade at the same price on both exchanges. When this condition does not hold, arbitrageurs come out of their lairs to hunt.

They buy the stock on the cheaper exchange and immediately sell it on the more expensive exchange, thus ensuring a virtually risk-free profit. These operations are performed at lightning speed, as the price differences do not last long due to the sheer power and speed of arbitrageurs. To clarify, here's an example:

- The stock's price at exchange A = \$10.00.
- The stock's price at exchange B = \$10.50.

The arbitrageur's algorithm in this case will do the following:

1. Buy the stock on exchange A for \$10.00.
2. Sell the stock immediately on exchange B for \$10.50.
3. Pocket the difference (\$0.50) and repeat until the gap is closed.



Trading and execution algorithms can be highly complex and require specialized knowledge and a certain market edge.

At this point, you should be aware of the two main uses of data science, data interpretation and data prediction:

Data interpretation

Also commonly referred to as *business intelligence* or simply *data intelligence*. The aim of deploying the algorithms is to understand the what and how of the data.

Data prediction

Also commonly referred to as *predictive analytics* or simply *forecasting*. The aim of deploying the algorithms is to understand the what's next of the data.

The main aim of using learning algorithms in financial markets is to predict future asset prices so that you can make an informed trading decision that results in capital appreciation at a success rate higher than random. I discuss many simple and complex learning algorithms in this book. These learning algorithms or models can be categorized as follows:

Supervised learning

Supervised learning algorithms are models that require labeled data to function. This means that you must provide data so that the model trains itself on these past values and understands the hidden patterns so that it can deliver future outputs when encountering new data. Examples of supervised learning include linear regression algorithms and random forest models.

Unsupervised learning

Unsupervised learning algorithms are models that do not require labeled data to function. This means that they can do the job with unlabeled data since they are built to find hidden patterns on their own. Examples include clustering algorithms and principal component analysis (PCA).

Reinforcement learning

Reinforcement learning algorithms are models that do not require data at all, as they discover their environment and learn from it on their own. In contrast to supervised and unsupervised learning models, reinforcement learning models gain knowledge through feedback obtained from the environment via a reward system. Since this is generally applied to situations in which an agent interacts with the environment and learns to adopt behaviors that maximize the reward over time, it may not be the go-to algorithm for time series regression. On the other hand, it can be used to develop a policy that can apply to time series data to create predictions.

As you may have noticed, the book's title is *Deep Learning for Finance*. This means that in addition to covering other learning models, I will be spending a sizable portion of the book discussing deep learning models for time series prediction. Deep learning mostly revolves around the use of neural networks, an algorithm discussed in depth in [Chapter 8](#).

Deep supervised learning models (such as deep neural networks) can learn hierarchical representations of the data because they include many layers, with each layer extracting features at a different level of abstraction. As a result, hidden and complex patterns are learned by deep models that may be difficult for shallow (not deep) models to learn.

On the other hand, shallow supervised learning models (like linear regression) have a limited ability to learn complex, nonlinear relationships. But they require less computational effort and are therefore faster.

Data science algorithms are deployed pretty much everywhere nowadays, not just in finance. Some applications include the following:

- *Business analytics*: Optimizing pricing, predicting customer turnover, or improving marketing initiatives using data analysis
- *Healthcare*: Improving patient outcomes, finding innovative therapies, or lowering healthcare costs through in-depth analysis of patient data
- *Sports*: Analyzing sports data to enhance team performance, player scouting, or bets
- *Research*: Analyzing data to support scientific investigation, prove theories, or gain new knowledge

When someone talks about data science applications, it helps to know what a data scientist does. A *data scientist* must evaluate and understand complex data in order to get insights and provide guidance for decision-making. Common tasks involved in this include developing statistical models, applying machine learning techniques, and visualizing data. They support the implementation of data-driven solutions and inform stakeholders of their results.



Data scientists are different from data engineers. Whereas a data scientist is concerned with the interpretation and analysis of data, a data engineer is concerned with the tools and infrastructure needed to gather, store, and analyze data.

By now you should understand everything you need to get started with data science. Let's introduce the second main topic of the book: financial markets.

Introduction to Financial Markets and Trading

The aim of this book is to present a hands-on approach to applying different learning models to forecast financial time series data. It is therefore imperative to gain solid knowledge of how trading and financial markets work.

Financial markets are places where people can trade financial instruments, such as stocks, bonds, and currencies. The act of buying and selling is referred to as *trading*. The main, but not only, aim of buying a financial instrument is capital appreciation. The buyer believes that the value of the instrument is greater than its price; therefore, the buyer buys the stock (*goes long*) and sells whenever they believe that the current price equals the current value. In contrast, traders can also make money if the price of the instrument goes down. This process is referred to as *short selling* and is common in certain markets such as futures and foreign exchange (FX).

The process of short selling entails borrowing the financial instrument from a third party, selling it on the market, and buying it back, before returning it to the third party. Ideally, as you expect the price of the instrument to drop, you would buy it back at a lower cost (after the price decrease) and give it back to the third party at the market price, thus pocketing the difference. The following examples explain these concepts further:

Long (buy) position example

A trader expects the price of Microsoft shares to increase over the next couple of years due to improved technology regulations, which would increase earnings. They buy a number of shares at \$250 and aim to sell them at \$500. The trader therefore has a long position on Microsoft stock (also referred to as being *bullish*).

Short (sell) position example

A trader expects the price of Lockheed Martin shares to decrease over the next couple of days due to signals from a technical strategy. They sell short a number of shares at \$450 and aim to buy them back at \$410. The trader therefore has a short position on Lockheed Martin stock (also referred to as being *bearish*).



Markets that are trending upward are referred to as bullish markets. Derived from the word *bull* and the bull's aggressive nature, being bullish is related to optimism, euphoria, and greed.

Markets that are trending downward are referred to as bearish markets. Derived from the word *bear* and its defensive nature, being bearish is related to pessimism, panic, and fear.

Financial instruments may come in their raw form (spot) or in a derivative form. *Derivatives* are products that traders use to trade markets in certain ways. For

example, a *forward* or a *futures* contract is a derivative contract where a buyer locks in a price for an asset to buy it at a later time.

Another type of derivative is an option. An *option* is the right, but not the obligation, to buy a certain asset at a specific price in the future by paying a premium now (the option's price). When a buyer wants to buy the underlying stock, they exercise their option to do so; otherwise, they may let the option expire.

Trading activity may also occur for hedging purposes, as it is not limited to just speculation. An example of this is Air France (the main French airline company) hedging its business operations by buying oil futures. Buying oil futures protects Air France from rising oil prices that may hurt its main operations (aviation). The rising costs from using fuel to power the planes are offset by the gains from the futures. This allows the airline to focus on its main business. This whole process is called *hedging*.

As another example, let's say an airline company expects to consume a certain amount of fuel in the next six months, but it is worried about the potential increase in oil prices over that period. To protect against this price risk, the airline can enter into a futures contract to purchase oil at a fixed price on a future date.

If the price of oil increases during that time, the airline would still be able to purchase the oil at the lower, fixed price agreed upon in the futures contract. If the price of oil decreases, the airline would be obligated to pay the higher, fixed price, but the lower market price for the oil would offset that cost.

In this way, the airline can mitigate the risk of price fluctuations in the oil market and stabilize its fuel costs. This can help the airline better manage its budget and forecast its future earnings. As you can see, the aim is not to make financial gains from the trading operations; it is simply to stabilize its costs by locking in a known price for oil.

Typically, financial instruments are grouped into asset classes based on their type:

Stock markets

A *stock market* is an exchange place (electronic or physical) where companies issue shares of stock to raise money for business. When people buy shares of a company's stock, they become part owners of that company and may become entitled to dividends according to the company's policy. Depending on the stock, they can also gain the right to vote in board meetings.

Fixed income

Governments and businesses can borrow money in the fixed income market. When a person purchases a bond, they are effectively lending money to the borrower, who has agreed to repay the loan along with interest. Depending on the borrower's creditworthiness and the prevailing interest rates, the bond's value may increase or decrease.

Currencies

The FX market, also referred to as the *currencies market*, is a place where people may purchase and sell various currencies. The value of a nation's currency can increase or decrease based on a variety of variables, including the economy, interest rates, and the nation's political stability.

Commodities

Agricultural products, gold, oil, and other physical assets with industrial or other uses are referred to as *commodities*. They typically offer a means to profit from global economic trends as well as being a form of hedge against inflation.

Alternative investments

In the world of finance, nontraditional investments such as real estate, private equity funds, and hedge funds are referred to as *alternative asset classes*. These asset classes have the potential to offer better returns than traditional assets and offer the benefit of diversity, but they also tend to be less liquid and may be more difficult to evaluate.

It's crucial to remember that each of these asset classes has unique qualities and various levels of risk, so investors should do their homework before investing in any of these assets.

Financial markets allow businesses and governments to raise the money they need to operate. They also allow investors to make money by speculating and investing in interesting opportunities. Trading activities provide liquidity to the markets. And the more liquid a market is, the easier and less costly it is to trade in it. But how do markets really work? What causes the prices to go up and down?

Market microstructure is the research that deals with the trading of securities in financial markets. It looks at how trading works as well as how traders, investors, and market makers behave. Understanding price formation and the variables that affect trading costs is the aim of market microstructure research.

Order flow, liquidity, market effectiveness, and price discovery are just a few of the many subjects covered by market microstructure research. Additionally, this research looks at how various trading techniques, including limit orders, market orders, and algorithmic trading, affect market dynamics. *Liquidity* is possibly the most important market microstructure concept. It describes how easily an asset may be bought or sold without materially changing its price. Liquidity can vary among financial instruments and over time. It can be impacted by a number of variables, including trading volume and volatility.

Finally, I want to discuss another important area of market microstructure: *price discovery*. This refers to the method used to set prices in a market. Prices can be affected by elements like order flow, market maker activity, and the presence of various trading methods.

Imagine you want to buy a sizable number of shares in two stocks: stock A and stock B. Stock A is very liquid, while stock B is very illiquid. If you want to execute the buy order on stock A, you are likely to get filled at the desired market price with minimal to no impact. However, with stock B, you are likely to get a worse price, as there are not enough sellers willing to sell at your desired buy price. Therefore, as you create more demand from your orders, the price rises to match the sellers' prices, and thus, you will buy at a higher (worse) price. This is the impact liquidity can have on your trading.

Applications of Data Science in Finance

Let's begin peeking into the main areas of data science for finance. Every field has its challenges and problems that need simple and complex solutions. Finance is no different. Recent years have seen a gigantic increase in the use of data science to improve the world of finance, from the corporate world to the markets world. Let's discuss some of these areas:

Forecasting the market's direction

The aim of using data science on financial time series is to uncover patterns, trends, and relationships in historical market data that can be used to make predictions about future market movements.

Financial fraud detection

Financial transactions can be examined for patterns and anomalies using data science models, which attempt to spot possible fraud. One way to use data science to stop financial fraud is to examine credit card transaction data for unusual or suspicious patterns of expenditures, such as numerous minor purchases made in quick succession or significant or frequent purchases made from the same store.

Risk management

Data science can be used to examine financial data and spot potential risks to portfolios. This can involve analyzing vast amounts of historical data using methods like statistical modeling, machine learning, and artificial intelligence to spot patterns and trends that can be used to forecast risk factors.

Credit scoring

Data science can be used to examine financial data and credit history, forecast a person's or a company's creditworthiness, and make loan decisions. Utilizing financial data, such as income and credit history, to forecast a person's creditworthiness is one example of applying data science for credit score research. This can involve using techniques such as statistical modeling and machine learning to develop a prediction model that can use a number of indicators, such as prior credit performance, income, and job history, to evaluate a person's likelihood of repaying a loan.

Natural language processing

To make better judgments, NLP analyzes and extracts insights from unstructured financial data, such as news articles, reports, and social media posts. NLP uses the sentiment of the text to extract possible trading opportunities stemming from the intentions and feelings of market participants and experts. NLP falls into the field of sentiment analysis (with help from machine learning).

Summary

The data science field keeps growing every day with the ongoing introduction of new techniques and models for improving data interpretation. This chapter provided a simple introduction to what you need to know about data science and how you can use it in finance.

The next three chapters present the knowledge in statistics, probability, and math that you may need when trying to understand data science models. Even though the aim of the book is to present a hands-on approach to creating and applying the different models using Python, it helps for you to understand what you're dealing with instead of blindly applying them to data.

If you need a Python refresher, see [Chapter 6](#), which is a basic introduction. It sets the foundation for what's to come next in the book. You do not need to become a Python master, but you must understand the code and what it refers to, and especially how to debug and detect errors in the code.

Essential Probabilistic Methods for Deep Learning

The rise and accessibility of technology have made it possible for everyone to deploy machine learning and deep learning algorithms for data analysis and optimization. But unfortunately, a large number of users do not understand the basics of the different learning models. This makes machine learning nothing short of a mystery box to them, which is a recipe for disaster.

Understanding fundamental concepts in probability, statistics, and math is essential for understanding and mastering data as well as for creating models that seek to interpret and forecast data. This chapter presents the basics of probability that are either directly or indirectly related to the algorithms. Note that you are unlikely to use these probability concepts in your everyday life, but it's important to know where some algorithms draw their assumptions from.

A Primer on Probability

Probability is all about describing random variables and random events. The world is filled with randomness, and the best way to find your way through chaos is to try to explain it using probabilistic methods. Granted, the phrase *explain chaos* may be an oxymoron, as chaos cannot really be explained, but we humans cannot relinquish control over uncertain events. This is why we have developed tools to make sense out of our scary world.

You may wonder what is the use of understanding the basics of probability when trying to develop machine learning algorithms for financial trading. This is a reasonable question, and you must know that the foundations of a discipline do not necessarily resemble it.

For example, to become a pilot you have to study aerodynamics, which is filled with technical concepts that do not resemble the final skill. This is similar to what is being done in this chapter; by studying probabilistic essentials, you give your brain a proper warm-up for what's to come.

Knowing the utility of what you are learning should give you a motivation boost. Here are some key probability topics that are important for machine learning:

Probability distribution functions

The possibility of seeing various outcomes of a random variable is described by a *probability distribution*. For many machine learning techniques, it is essential to comprehend the features and attributes of typical probability distributions. Probability distribution functions also describe different types of time series data, which in turn helps in choosing the right algorithm. For simplicity and coherence, this topic is discussed in [Chapter 3](#).

Hypothesis testing

Hypothesis testing is used to establish whether a population-based assertion is more likely to be correct or incorrect based on a sample of data. Stationarity tests use hypothesis testing and are discussed in [Chapter 3](#).

Decision trees

Decision trees are a type of machine learning algorithm that borrows from probabilistic concepts such as conditional probability, a concept covered in this chapter. For more detail on decision trees, see [Chapter 7](#).

Information theory

Information theory is the complex study of how information is quantified, stored, and transmitted. It is incorporated into numerous machine learning techniques, including decision trees. It is also used in a type of nonlinear correlation measure called the maximal information coefficient, which is discussed in [Chapter 3](#).

Introduction to Probabilistic Concepts

The most basic piece of probabilistic information is a *random variable*, which is an uncertain number or outcome. Random variables are used to model events that are considered uncertain, such as the future return of a currency pair.

A random variable is either discrete or continuous. A *discrete random variable* has a finite set of values, while a *continuous random variable* has values within a certain interval. Consider the following examples to clarify things:

- An example of a discrete random variable would be the result of rolling a die. The outcomes are limited by the following set: {1, 2, 3, 4, 5, 6}.
- An example of a continuous random variable would be the daily price returns of EURUSD (the exchange rate of one euro expressed in US dollars).

Random variables are described by *probability distributions*, which are functions that give the probability of every possible value of these random variables. Generally, a histogram is used to show the probability. Histogram plotting is discussed in [Chapter 3](#).

At any moment, the probability that a certain event will unfold is between 0 and 1. This means that probability is assigned to random variables on a scale between 0 and 1 such that a probability of 0 represents zero chance of occurrence and a probability of 1 represents a certainty of occurrence.

You can also think of this in percentage terms, which range from 0% to 100%. Values within the two numbers are valid, which means that you can have a 0.5133 (51.33%) probability of a certain event occurring. Consider rolling a die that has six sides. What is the probability of getting a 3 knowing that the die is not manipulated in any way?

As the die has six sides, there are six equal probabilities for every outcome, which means that for any outcome, the probability is found as follows:

$$P(x) = \frac{1}{6} = 0.167$$

with $P(x)$ designating the probability of event x . This gives the answer to the question:

$$P(3) = \frac{1}{6} = 0.167$$

When a die is rolled, there can only be one result. It cannot give a 3 and a 4 simultaneously, since one side has to dominate the other. This is the concept of *mutual exclusivity*. Mutually exclusive events (such as getting a 3 or getting a 4 in a die roll) eventually sum to 1.

Take a look at the following example:

$$P(1) = \frac{1}{6} = 0.167$$

$$P(2) = \frac{1}{6} = 0.167$$

$$P(3) = \frac{1}{6} = 0.167$$

$$P(4) = \frac{1}{6} = 0.167$$

$$P(5) = \frac{1}{6} = 0.167$$

$$P(6) = \frac{1}{6} = 0.167$$

Summing all these mutually exclusive events gives 1, which means that the sum of the possible probabilities in a six-sided die is as follows:

$$P(1) + P(2) + P(3) + P(4) + P(5) + P(6) = 1$$



Stating that a random variable has a 0.8 probability of occurring is the same as stating that the same variable has a 0.2 probability of not occurring.

Probability measures can be conditional or unconditional. A *conditional probability* is when the occurrence of an event impacts the probability that another event occurs. For example, the probability of a sovereign interest rate hike given positive employment data is an example of a conditional probability. The probability of event A given the occurrence of event B is denoted by the mathematical notation $P(A|B)$.

In contrast, *unconditional probability* is not dependent on other events. Taking the example of conditional probability, you can formulate an unconditional probability calculation that measures the probability of an interest rate hike regardless of other economic events.

Probabilities have specific addition and multiplication rules with their own interpretations. Let's take a look at the formulas before seeing an example. The *joint probability* of the realization of two events is the probability that they will both occur. It is calculated using the following formula:

$$P(AB) = P(A|B) \times P(B)$$

That formula says the probability of occurrence for both A and B is the probability that A occurs given B occurs multiplied by the probability that B occurs. Therefore, the right side of the equation multiplies a conditional probability by an unconditional probability.

The *addition rule* is used to determine the probability that at least one of the two outcomes will occur. This works in two ways: one deals with mutually exclusive events, and the other deals with events that are not mutually exclusive.

If the events are not mutually exclusive, then to avoid double counting, the formula is:

$$P(A \text{ or } B) = P(A) + P(B) - P(AB)$$

If the events are mutually exclusive, then the formula is simplified to the following:

$$P(AB) = 0$$

$$P(A \text{ or } B) = P(A) + P(B) - 0$$

$$P(A \text{ or } B) = P(A) + P(B)$$

Notice how in mutually exclusive events, it's either A or B that can be realized, and therefore, the probability that both of them will occur is zero. To understand why you need to subtract the joint probability of A and B, take a look at [Figure 2-1](#).

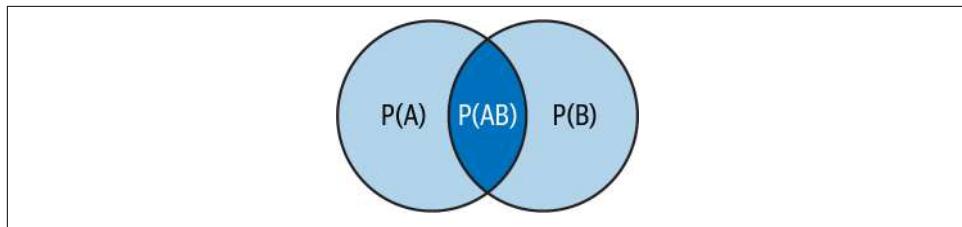


Figure 2-1. The addition rule of probability

Notice how the probability of either A or B occurring while they are mutually exclusive must not include their joint probability. Let's now look at the concept of independent events.

Independent events are not tied to one another (e.g., rolling the die twice). In this case, the joint probability is calculated as follows:

$$P(AB) = P(A) \times P(B)$$

Independent events therefore refer to instances where the occurrence of one event has absolutely zero impact on the occurrence of the other event(s). Let's see an example to validate this concept. Consider a simple coin toss. The probability of getting heads does not depend on what you got in the previous coin toss. Therefore, the probability of getting heads is always 0.50 (50%). To take things further, what is the probability of getting only heads after five coin tosses?

As the probability of each event is independent from the previous or the next one, the formula is as follows:

$$P(x) = 0.50 \times 0.50 \times 0.50 \times 0.50 \times 0.50 = 0.03125 = 3.125\%$$

The *expected value* of a random variable is the weighted average of the different outcomes. Therefore, the expected value is really another way of referring to the mean. Mathematically, the expected value is as follows:

$$E(X) = \sum_{i=1}^n (P(x_i)x_i)$$

Take a look at [Table 2-1](#) and try to calculate the expected value of the next employment numbers in a certain month of the year.

Table 2-1. Employment numbers

Nonfarm payrolls	Probability
300,000	0.1
400,000	0.3
500,000	0.5
600,000	0.1

Nonfarm payrolls refer to a monthly report issued by the US Department of Labor that gives information on the total number of paid employees in the nation, excluding those employed in the agriculture sector, as well as those employed by the government and nonprofit organizations.

From [Table 2-1](#), economists assume there is a 50% probability that there will be a 500,000 increase in the total number of paid employees and a 30% probability that there will be a 400,000 increase in the total number of paid employees. The expected value is therefore:

$$\begin{aligned} E(X) &= (300,000 \times 0.1) + (400,000 \times 0.3) + (500,000 \times 0.5) + (600,000 \times 0.1) \\ &= 460,000 \end{aligned}$$

Therefore, the number that represents the economists' consensus is 460,000, as it is the closest weighted value to most forecasts. It is the value that represents the dataset.



The main takeaways from this section are as follows:

- Probability describes random variables and random events. It is a value between 0 and 1.
- Probabilities of events may be grouped together to form more complex scenarios.
- The expected outcome is the weighted average of every probability in the designated universe.

Sampling and Hypothesis Testing

When populations are large, representative samples are taken so that they become the main describers of data. Take the United States. Its democratic system means that the people hold the right to decide their own fate, but it's not possible to go to every person and ask them for their detailed opinions on every topic out there. This is why elections are held and representatives are elected to act in the people's name.

Sampling refers to the act of selecting samples of data within a larger population and making conclusions about the statistical properties of the population. There are a few different methods of sampling. The best-known ones are the following:

Simple random sampling

With simple random sampling, each element in the population has an equal chance of being selected for the sample. This can be a random number generated on a labeled population where each individual has the same probability of being selected.

Stratified sampling

With stratified sampling, the population is divided into groups based on some characteristic, and then a simple random sample is taken from each group in proportion to its size.

Cluster sampling

With cluster sampling, the population is divided into clusters, and a random sample of clusters is selected. Then, all elements within the selected clusters are included in the sample.

Systematic sampling

With systematic sampling, an element is selected by choosing every n th individual from the population, where n is a fixed number. This means that it is not random but specified in advance.

A rule of thumb is that the more data you acquire, the better the metrics reflect the population. Sampling is extremely important in the world of machine learning, as quite often you are taking samples of data to represent the true population. For example, when performing a backtest on a trading strategy, you will be required to split the whole dataset into a *training sample* and a *testing sample* where the first is the sample of data on which the algorithm understands its structure (also known as the *in-sample set*), and the second is the sample of data on which the algorithm tests its predictive power (also known as the *out-of-sample set*).

Another example of using sampling is *cross validation*. With this technique, a dataset is divided into two or more subsets. The model is trained using one subset, and its results are tested using the other subset(s). For various subsets of the data, this procedure is repeated numerous times, and then the model's average performance is determined.

These terms are discussed in more depth in the coming chapters. For now, you should understand that the concept of sampling is very important in machine learning.

Sampling is not perfect, and errors may be possible, just as in any other estimation method. *Sampling error* refers to the difference between the statistic of the sample and the statistic of the population (if it's known). A *statistic* is a metric that describes the analyzed dataset (an example of this would be the mean, a statistic you will see in greater detail in [Chapter 3](#)). Now, what is the minimum sample size you should have to be able to make inferences about the population? The rule of thumb is to have a minimum of 30 observations, and the more the merrier. This brings the discussion to the *central limit theorem*, which states that random samples drawn from a population will approach a normal distribution (a probability distribution that is symmetric and bell shaped) as the sample gets larger.

The central limit theorem makes it simple to apply inferences and conclusions as hypothesis testing goes well with a normal distribution. Before proceeding to hypothesis testing, let's look at *confidence intervals*, which are ranges of values where the population parameter is expected to be. Confidence intervals are generally constructed by adding or subtracting a factor from the point estimate. For example, given a sample mean \bar{x} , a confidence interval can be constructed as follows:

$$\bar{x} \pm (\text{reliability factor} \times \text{standard error})$$

Let's try to understand the calculation step by step. The sample mean is an estimate of the population and is calculated because it is not possible to calculate the population mean. Therefore, by performing a random sample, the assumption is that the sample mean should be equal to the population mean. However, in real life, things may differ, and this is why you should construct a confidence interval using probabilistic methods.



The *significance level* is the threshold of the confidence interval. For example, a confidence interval of 95% means that with 95% confidence, the estimate should lie within a certain range. The remaining 5% probability that it does not is the significance level (generally marked with the letter alpha, α).

A *reliability factor* is a statistical measure that depends on the distribution of the estimate and the probability that it falls within the confidence interval. For the sake of simplicity, let's assume that the variance of the population is normal and the population is normally distributed. For a significance level of 5% (thus, a confidence interval of 95%), the reliability factor is 1.96 in this case (the way you get this number is less relevant to the discussion).

The *standard error* is the standard deviation of the sample. *Standard deviation* is discussed in greater depth in [Chapter 3](#); for now, just know that it represents the degree of fluctuation of the different values around the mean. Standard error is found using the following formula:

$$s = \frac{\sigma}{\sqrt{n}}$$

σ is the population standard deviation

\sqrt{n} is the square root of the population number

It is also worth knowing that for a 1% significance level, the reliability factor is 2.575, and for a 10% significance level, the reliability factor is 1.645. Let's take a look at a practical example to make sense of all this math.

Consider a population of 100 financial instruments (bonds, currency pairs, stocks, structured products, etc.). The mean annual return of these instruments is 1.4%. Assuming a population standard deviation of 4.34%, what is the confidence interval at a 1% significance level (99% confidence interval) of the mean?

The answer is determined by just plugging the values into the formula as follows:

$$1.4\% \pm 2.575 \times \frac{4.34\%}{\sqrt{100}} = 1.4\% \pm 1.11\%$$

This means that the confidence interval is between (0.29%, 2.51%).



If the sample size is small and/or the population standard deviation is unknown, a t-distribution may be a better choice than a normal distribution.

The *t-distribution* is a type of probability distribution used to model the distribution of a sample mean when the sample size is small and/or when the population standard deviation is unknown. It resembles the normal distribution in shape but with heavier tails, which represents the uncertainty associated with smaller sample sizes.

The next stop is hypothesis testing, a key probabilistic technique of getting conclusions on samples of data. This part is extremely important, as it's used in a lot of statistical analyses and models.

In statistics, *hypothesis testing* is a technique for drawing conclusions about a population from a small sample of data. It entails developing two competing hypotheses, the *null hypothesis* and the *alternative hypothesis*, about a population parameter and then figuring out which is more likely to be accurate using sample data.

For example, say that a financial analyst is evaluating two portfolios from a risk perspective. They formulate two hypotheses:

- The null hypothesis states that there is no significant difference in the volatility of the two portfolios.
- The alternative hypothesis states that there is a significant difference in the volatility of the two portfolios.

The hypothesis is then tested using statistical analysis to determine whether the difference in volatility is statistically significant or due to pure chance.

Following the definition of null and alternative hypotheses, a test statistic is computed using the sample data. To assess the result's significance, the test statistic is then compared to a critical value drawn from a standard distribution. The null hypothesis is rejected and the alternative hypothesis is accepted if the test statistic is inside the crucial zone. The null hypothesis is not rejected and the conclusion that there is insufficient evidence to support the alternative hypothesis is reached if the test statistic does not fall inside the crucial zone.

This is all a fancy way of saying that hypothesis testing basically involves creating two opposing scenarios, running a probability check, and then deciding which scenario is more likely true. Hypothesis testing can take two forms:

One-tailed test

An example of this would be to test if the return on certain financial instruments is greater than zero.

Two-tailed test

An example of this would be to test if the return on certain financial instruments is different from zero (meaning that it can be either greater than or less than zero). Hypothesis tests are generally two-tailed tests.

The null hypothesis is the one that you want to reject and therefore is tested in the hopes of getting rejected and accepting the alternative scenario. A two-tailed test takes the following general form:

$$H_0: x = x_0$$

$$H_a: x \neq x_0$$

As the alternative scenario allows for values above and below zero (which is the stated level in the null hypothesis), there should be two critical values. Therefore, the rule of a two-tailed test is to reject the null hypothesis if the test statistic is greater than the upper critical value or less than the lower critical value. For instance, for a normally distributed dataset, the test statistic is compared with the critical values (at 5% significance level) at +1.96 and -1.96. The null hypothesis is rejected if the test statistic falls outside the range of +1.96 and -1.96.

The process of hypothesis testing entails the calculation of the test statistic. This is done by comparing the point estimate of the population parameter with the hypothesized value of the null hypothesis. Both are then scaled by the standard error of the sample. The mathematical representation is as follows:

$$\text{Test statistic} = \frac{\text{Sample statistic} - \text{Hypothesized value}}{\text{Standard error}}$$

An important consideration in hypothesis testing is that the sample may not be representative, which leads to errors in describing the population. This gives rise to two types of errors:

Type I error

This error occurs when rejecting the null hypothesis even though it is true.

Type II error

This error occurs when failing to reject the null hypothesis even though it is false.

Intuitively, the significance level is the probability of making a type I error. Remember that if $\alpha = 5\%$, then there is a 5% chance of rejecting a true null hypothesis by mistake. An example would make things clearer.

Consider an analyst doing research on the annual returns of a long-short portfolio over a period of 20 years. The mean annual return was 1% with a standard deviation of 2%. The analyst's opinion is that the mean annual return is not equal to zero, and they want to construct a 95% confidence interval for this and then construct a hypothesis test. You would proceed as follows:

1. State the variables. The size of the sample is 20, the standard deviation is 2%, and the mean is 1%.
2. Calculate the standard error, which in this case is 0.44% as per the formula.
3. Define the critical values for the 95% confidence interval. The critical values are +1.96 and -1.96. To find the confidence interval, add and subtract the margin of error from the sample mean. The confidence interval is therefore (0.13%, 1.86%).

4. Specify the null hypothesis, which is, according to the analyst's opinion, a two-tailed test. The null hypothesis is that the annual return equals zero. You should reject it if the test statistic is less than -1.96 or greater than $+1.96$.
5. Using the formula to find the test statistic gives 2.27 . Therefore, the null hypothesis is rejected.

One more important metric to discuss is the *p-value*. The *p-value* is the probability of seeing a test statistic more extreme than the one seen in the statistical test given that the null hypothesis is true. Comparing a *p-value* to a significance level—typically 0.05 —allows you to understand it. The result is deemed statistically significant, and the null hypothesis is rejected in favor of the alternative hypothesis if the *p-value* is less than or equal to the significance level.

If the *p-value* is less than the significance level of 5% , it means that there is a 5% chance you will see a test statistic as extreme as the current one if the null hypothesis is true. Another way of defining the *p-value* is to consider it as being the smallest significance level for which the null hypothesis can be rejected.



The main takeaways from this section are as follows:

- Sampling refers to the collection of data within a population, with the aim of making conclusions about the statistical properties of the aforementioned population.
- Hypothesis testing is a technique for drawing conclusions about a population from a small sample of data.

A Primer on Information Theory

Information theory is a complex field in abstract mathematics that is closely related to probability. It is the study of how information is quantified, stored, and transmitted. There are three conditions of occurrence when it comes to an event:

Uncertainty

If the event has not occurred yet

Surprise

If the event has just occurred

Information

If the event has occurred in the past

One of the key concepts in information theory is *entropy*, which is the level of uncertainty or randomness in a message or information source and describes the degree to which an event or message is unexpected. In contrast, *information gain* measures the reduction in entropy (surprise) when receiving new information.

Basically, information theory describes the surprise of events. When an event has a low probability of occurrence, it has more surprise and hence more information to provide. Similarly, when an event has a high probability of occurrence, it has less surprise and therefore less information. What you should retain from this is that the amount of information learned from an unlikely event is greater than the amount of information learned from a likely event.

Before starting to dig a little deeper into the field of information theory, it is important to understand what a logarithm is and, for that matter, what an exponent is. A general exponential function takes a certain constant or a variable to a certain power:

$$f(x) = a^x$$

In other words, the *exponent of a number* is the number of times you will multiply it by itself:

$$4^3 = 4 \times 4 \times 4 = 64$$

A logarithm is the opposite of an exponent, and its aim is to find the exponent—knowing 4 and 64 from the previous example and finding 3:

$$\log_4(64) = 3$$

A logarithm therefore is the answer to how many of one number to multiply to get another number. Since they are literally inverse functions, you can use them together to simplify or even solve for x . Take the following example:

$$\log_4(x) = 3$$

The objective here is to find x given the logarithmic function. The first step is simply to use the exponential function on one side as you want it to cancel out the logarithm on the right (inverse functions cancel each other out). This gives us the following result:

$$4^{\log_4(x)} = 4^3$$

$$x = 4^3$$

$$x = 64$$

Logarithms can have different bases. However, the most used logarithm has a base of 10. In computer science, base 2 logarithms represent bits (binary digits). Therefore, information is represented as bits. The formula of information gain is as follows:

$$H(x_i) = -\log_2(P(x_i))$$

Let's assume two variables x and y , where x has a probability of 1 (100% and therefore certain) and y has a probability of 0.5 (50% and therefore mostly random). What would be the information value in these two cases? The answer is as follows:

$$H(x) = -\log_2(P(1)) = 0$$

$$H(y) = -\log_2(P(0.5)) = 1$$

So the certain event has an information value of zero, and the one that has a 50-50 chance of realizing has an information value of 1. What about the very unlikely event z that has a probability of 0.05 (5%)?

$$H(z) = -\log_2(P(0.05)) = 4.32$$

A negative relationship between probability and information is one of the principles of information theory. Entropy and information are related concepts, but they have different meanings and applications.

Entropy is a metric used to assess how chaotic or random a system is. Entropy describes how uncertain or unpredictable a signal is. The degree of disorder or unpredictability in the system or communication increases as entropy increases.

Information is the decrease in entropy or uncertainty that happens as a result of receiving a signal. A signal's ability to lessen the receiver's uncertainty or entropy increases with its informational content.



Entropy is maximized whenever all the events are equally likely.

Entropy is calculated using the following formula:

$$S(x_n) = \sum_{i=1}^n (-\log_2(P(x_i)).(P(x_i)))$$

Therefore, entropy is the average of the sum of logarithms times their respective probabilities.

Now let's discuss the final concept of the section, *information gain*. The reduction in entropy caused by changing a dataset is calculated via information gain.



Information gain is one of the key concepts you will see in [Chapter 7](#) with decision trees, and therefore, you may want to refer back to this section after reading that chapter.

The typical way to calculate information gain is by comparing the entropy of a dataset before and after a transformation. Recall that entropy is maximized when all the outcomes of a random event have the same probability. This can also be presented as a distribution, where a symmetrical distribution (such as the normal distribution) has high entropy and a skewed distribution has low entropy.



Minimizing entropy is related to maximizing information gain.

Before closing this introductory section on information theory, let's look at the concept of *mutual information*. This measure is calculated between two variables, hence the name *mutual*, and it measures the reduction in uncertainty of a variable given another variable. The formula for mutual information is as follows:

$$MI(x, y) = S(x) - S(x|y)$$

$MI(x, y)$ is the mutual information of x and y

$S(x)$ is the entropy of x

$S(x|y)$ is the conditional entropy of x and y

Mutual information therefore measures the dependence between variables. The greater the mutual information, the bigger the relationship between the variables (a value of zero represents independent variables). Keep this concept in mind, as you will see it in "[Correlation](#)" on page 63. This is because mutual information can also be a measure of nonlinear correlation between variables.



To summarize, here is what you need to retain in information theory to have a basic knowledge of what's to come:

- Information theory uses concepts from probability to calculate information and entropy, which are used in machine learning models and other calculations (such as correlation).
- Information is the decrease in entropy or uncertainty that happens as a result of receiving a signal. Entropy is a metric used to assess how chaotic or random a system is.
- Mutual information is a measure of dependence between two random variables. It can also be used to calculate the correlation between the two.
- Tools from information theory are used in some machine learning models such as decision trees.

Summary

An understanding of probability presents a basic framework before moving to more advanced topics. This chapter skimmed over the concepts that you may encounter when dealing with machine and deep learning models. It is important to understand how probability is calculated and how hypothesis testing is performed (even though, in reality, algorithms will do this for you).

The next chapter is extremely important and presents the statistical knowledge you need, not just for machine learning but also for financial trading and even complex data analysis.

Descriptive Statistics and Data Analysis

Descriptive statistics is a field that describes data and extracts as much information as possible from it. Basically, descriptive statistics can act like the representative of the data since it summarizes its tendencies, behavior, and trends.

Trading and analysis borrows a lot from the metrics used in descriptive statistics. This chapter covers the main concepts of descriptive statistics and data analysis. I always found that the best educational tools are practical examples, so I will explain these concepts using an example of an economic time series, the consumer price index (CPI).

The CPI measures the prices paid monthly by urban consumers for a selection of products and services; every month a new observation is released to the public, thus forming a continuous time series. The inflation rate between any two time periods is measured by percentage changes in the price index. For example, if the price of bread last year was \$1.00 and the price today is \$1.01, then the inflation is 1.00%. The CPI is typically released on a year-on-year basis, which means that it is reported as the difference between the current monthly observation and the observation 12 months ago.

Import the CPI data as follows:

```
# Importing the required library
import pandas_datareader as pdr
# Setting the beginning and end of the historical data
start_date = '1950-01-01'
end_date   = '2023-01-23'
# Creating a dataframe and downloading the CPI data
cpi = pdr.DataReader('CPIAUCSL', 'fred', start_date, end_date)
# Printing the latest five observations of the dataframe
print(cpi.tail())
# Checking if there are nan values in the CPI dataframe
count_nan = cpi['CPIAUCSL'].isnull().sum()
# Printing the result
print('Number of nan values in the CPI dataframe: ' + str(count_nan))
```

```

# Transforming the CPI into a year-on-year measure
cpi = cpi.pct_change(periods = 12, axis = 0) * 100
# Dropping the nan values from the rows
cpi = cpi.dropna()

```



You can download all the code samples throughout the book from the book's dedicated [GitHub page](#).

The year-on-year change is the most observed transformation on the CPI, as it gives a clear and simple measurement of the change in the overall price level over a sufficient period of time to account for short-term swings and seasonal impacts.

Hence, the yearly change of the CPI serves as a gauge of the general trend in inflation. It is also simple to comprehend and compare across other nations and historical times, making it a popular measure among policymakers and economists (despite the flaw of element weightings in the baskets between countries). The following sections show how to make statistical sense of time series data using the CPI example.

Measures of Central Tendency

Central tendency refers to the metrics that summarize the dataset into a value that can represent them. The best-known central tendency measure is the mean (average). The *mean* is simply the sum of the values divided by their quantity. It is the value that represents the data the best. The mathematical formula of the mean is as follows:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} (x_1 + \dots + x_n)$$

Let's take a simple example of two datasets. Suppose you want to calculate the mean on dataset A and dataset B. How would you do it?

- Dataset A = [1, 2, 3, 4, 5]
- Dataset B = [1, 1, 1, 1]

Dataset A contains five values (quantity) with a total sum of 15. Using the preceding formula, the mean is equal to 3. Dataset B contains four values with a total sum of 4. This means that the mean is equal to 1.

[Figure 3-1](#) shows the US CPI year-on-year values since 2003. The higher dashed line is the monthly mean calculated since 2003. The lower dashed line symbolizes zero, and below it are deflationary periods.



When all the values in a dataset are the same, the mean is the same as the values.

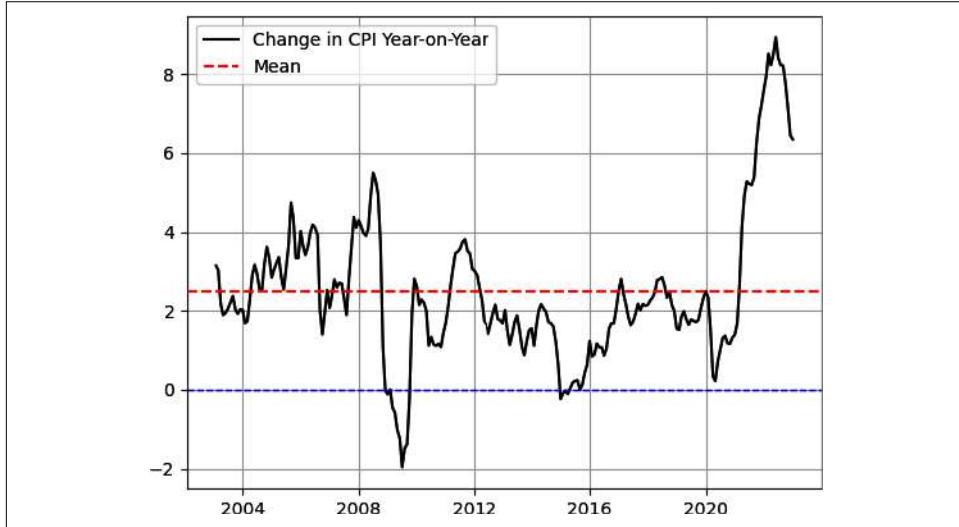


Figure 3-1. US CPI year-on-year values since 2003, with the higher dashed line representing the mean

You can create Figure 3-1 by using the following code:

```
# Calculating the mean of the CPI over the last 20 years
cpi_latest = cpi.iloc[-240:]
mean = cpi_latest["CPIAUCSL"].mean()
# Printing the result
print('The mean of the dataset: ' + str(mean), '%')
# Importing the required library
import matplotlib.pyplot as plt
# Plotting the latest observations in black with a label
plt.plot(cpi_latest[:,], color = 'black', linewidth = 1.5,
          label = 'Change in CPI Year-on-Year')
# Plotting horizontal lines that represent the mean and the zero threshold
plt.axhline(y = mean, color = 'red', linestyle = 'dashed',
            label = 'Mean')
plt.axhline(y = 0, color = 'blue', linestyle = 'dashed', linewidth = 1)
plt.grid()
plt.legend()
```

The output of the mean should be as follows.

The mean of the dataset: 2.49 %

This means that the average yearly inflation is around 2.50%. Even though the Federal Reserve does not have a clearly defined inflation target, it is generally believed that there is a consensus to maintain the annual change in inflation at around 2.00%, which is not far from the historical observations. With the high inflation numbers recorded since 2021 as a result of political and economic turmoil, it becomes necessary to revert back to the mean to stabilize the current situation. This example gives a numerical value to what is referred to as normality (~2.50%) since 2003.

Clearly, with the high inflation numbers (~6.00%) around the beginning of 2023, the situation is a bit far from normality, but how far? This question is answered in the next section, which discusses measures of variability. For now, let's continue the discussion on central tendency.

The next measure is the *median*, which in simple terms is the value that splits the dataset into two equal sides. In other words, if you arrange the dataset in an ascending order, the middle value is the median. The median is used whenever there are many outliers or there is skew in the distribution (which may bias the mean and make it less representative).

There are generally two topics associated with calculating the median. The first one relates to a dataset that contains an even number of values (e.g., 24 observations), and the second one relates to a dataset that contains an uneven number of values (e.g., 47 observations):

Calculating the median of an even dataset

If the arranged dataset has an even number of values, the median is the average of the two middle values.

Calculating the median of an uneven dataset

If the arranged dataset has an uneven (odd) number of values, the median is simply the middle value.

Let's take a simple example of two datasets. Suppose you want to calculate the median on dataset A and dataset B. How would you do it?

- Dataset A = [1, 2, 3, 4, 5]
- Dataset B = [1, 2, 3, 4]

Dataset A contains five values, which is an uneven number. This means that the middle value is the median. In this case, it is 3 (notice how it is also the mean of the dataset). Dataset B contains four values, which is an even number. This means that the average between the two middle values is the median. In this case, it is 2.5, which is the average between 2 and 3.

Figure 3-2 shows the US CPI year-on-year values since 2003. The higher dashed line is the monthly median calculated since 2003. The lower dashed line symbolizes zero. Basically, this is like **Figure 3-1**, but instead of the mean, the median is plotted.

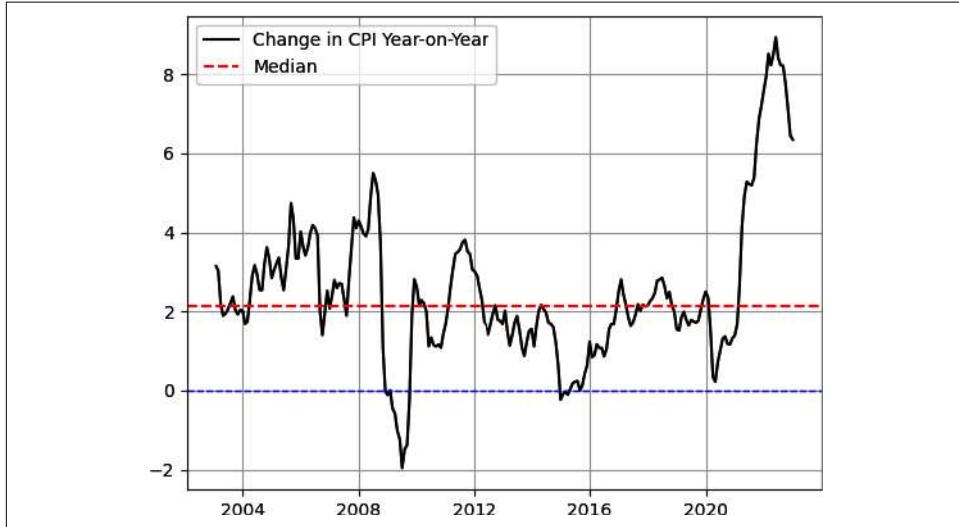


Figure 3-2. US CPI year-on-year values since 2003, with the higher dashed line representing the median

You can create **Figure 3-2** by using the following code:

```
# Calculating the median
median = cpi_latest["CPIAUCSL"].median()
# Printing the result
print('The median of the dataset: ' + str(median), '%')
# Plotting the latest observations in black with a label
plt.plot(cpi_latest[:,], color = 'black', linewidth = 1.5,
          label = 'Change in CPI Year-on-Year')
# Plotting horizontal lines that represent the median and the zero threshold
plt.axhline(y = median, color = 'red', linestyle = 'dashed',
            label = 'Median')
plt.axhline(y = 0, color = 'blue', linestyle = 'dashed', linewidth = 1)
plt.grid()
plt.legend()
```

The output of the median should be as follows:

`The median of the dataset: 2.12 %`

Clearly, the median is less impacted by the recent outliers that are coming from unusual environments. The median is around 2.12%, which is more in line with the implied target of 2.00%.



Remember that [Chapter 6](#) will give you all you need to know about the Python snippets you are seeing in this chapter, so you don't need to worry if you are missing out on the coding concepts.

The last central tendency measure in this section is the *mode*, which is the value that is the most frequently observed (but also the least used in data analysis).

Let's take a simple example of two datasets. Suppose you want to calculate the mode on the following datasets. How would you do it?

- Dataset A = [1, 2, 2, 4, 5]
- Dataset B = [1, 2, 3, 4]
- Dataset C = [1, 1, 2, 2, 3]

Dataset A contains two times the value 2, which makes it the mode. Dataset B doesn't have a mode, as every value is observed once. Dataset C is *multimodal* since it contains multiple modes (which are 1 and 2).



The mode is useful with categorical variables (like credit rankings) as opposed to continuous variables (like price and returns time series).

You are unlikely to use the mode to analyze time series, as the mean and the median are more useful. Here are a few examples that use the mean and the median in financial analysis:

- Calculating a moving mean (average) on price data to detect an underlying trend. You will see more about moving averages in [Chapter 5](#).
- Calculating a rolling median on a price-derived indicator to detect its neutrality zone.
- Calculating the expected return of a security using the historical mean.

Central tendency metrics are important to understand, especially since the mean and the median are heavily used not only as standalone indicators but also as components in more complex measures.



The key takeaways from this section are as follows:

- There are mainly three central tendency measures: the mean, the median, and the mode.
- The mean is the sum divided by the quantity, while the median is the value that splits the data in half. The mode is the value that occurs most frequently in the dataset.

Measures of Variability

Measures of variability describe how spread out the values in a dataset are relative to the central tendency measures (mostly the mean). The best-known measure of variability is the variance.

The *variance* describes the variability of a set of numbers from their mean. The idea behind the variance's formula is to determine how far away from the mean each data point is, then to square those deviations to make sure all the numbers are positive (this is because distance cannot be negative), and finally to divide the deviations by the number of observations.

The formula to find the variance is as follows:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

This formula calculates the sum of the squared deviations of each data point from the mean, thus giving different distance observations, and then calculates the mean of these distance observations.

Let's take a simple example of two datasets. Suppose you want to calculate the variance of dataset A and dataset B. How would you do it?

- Dataset A = [1, 2, 3, 4, 5]
- Dataset B = [5, 5, 5, 5]

The first step is to calculate the mean of the dataset as that is the benchmark from where you will calculate the variability of the data. Dataset A has a mean of 3. The next step calculates the variance:

$$(x_1 - \bar{x})^2 = (1 - 3)^2 = 4$$

$$(x_2 - \bar{x})^2 = (2 - 3)^2 = 1$$

$$(x_3 - \bar{x})^2 = (3 - 3)^2 = 0$$

$$(x_4 - \bar{x})^2 = (4 - 3)^2 = 1$$

$$(x_5 - \bar{x})^2 = (5 - 3)^2 = 4$$

The previous results are summed as follows:

$$4 + 1 + 0 + 1 + 4 = 10$$

And finally, the result is divided by the quantity of the observations to find the variance:

$$\sigma^2 = \frac{10}{5} = 2$$

As for dataset B, you should think about it intuitively. If the observations are all equal, they all represent the dataset, which also means that they are their own mean. What would you say about the variance of the data in this case, considering that all the values are equal to the mean?

If your response is that the variance is zero, then you are correct. Mathematically, you can calculate it as follows:

$$(x_1 - \bar{x})^2 = (5 - 5)^2 = 0$$

$$(x_2 - \bar{x})^2 = (5 - 5)^2 = 0$$

$$(x_3 - \bar{x})^2 = (5 - 5)^2 = 0$$

$$(x_4 - \bar{x})^2 = (5 - 5)^2 = 0$$

The previous results sum to zero, and if you divide zero by 4 (the quantity of the dataset), you will get zero. Intuitively, there is no variance because all the values are constant and they do not deviate from their mean:

$$\sigma^2 = \frac{0}{4} = 0$$

Remaining in the inflation example, you can calculate the variance using the following code:

```
# Calculating the variance
variance = cpi_latest["CPIAUCSL"].var()
# Printing the result
print('The variance of the dataset: ' + str(variance), '%')
```

The output of the variance should be as follows:

The variance of the dataset: 3.62 %

There is a flaw nonetheless, and it is that the variance represents squared values that are not comparable to the mean since they use different units. This is easily fixed by taking the square root of the variance. Doing so brings the next measure of variability, the *standard deviation*. It is the square root of the variance and is the average deviation of the values from the mean.

A low standard deviation indicates that the values tend to be close to the mean (low volatility), while a high standard deviation indicates that the values are spread out over a wider range relative to their mean (high volatility).



The terms *standard deviation* and *volatility* are used interchangeably. They refer to the same thing.

The formula to find the standard deviation is as follows:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$$

If you consider the previous examples with the variance, then the standard deviation can be found as follows:

$$\sigma_{\text{Dataset A}} = \sqrt{2} = 1.41$$

$$\sigma_{\text{Dataset B}} = \sqrt{0} = 0$$

Standard deviation is commonly used with the mean since they both use the same units. You will soon understand the importance of this stat when I discuss the normal distribution function in the next section.

You can calculate the standard deviation in Python using the following code:

```
# Calculating the standard deviation
standard_deviation = cpi_latest["CPIAUCSL"].std()
# Printing the result
print('The standard deviation of the dataset: ' +
      str(standard_deviation), '%')
```

The output of the standard deviation should be as follows:

```
The standard deviation of the dataset: 1.90 %
```

How are you supposed to interpret the standard deviation? On average, the CPI year-on-year values tend to be $\pm 1.90\%$ from the mean of the same period, which is at 2.49%. In “[Measures of Shape](#)” on page 45, you will see how to make better use of standard deviation figures.

The last measure of variability discussed in this section is the range. The *range* is a very simple stat that shows the distance between the highest value and the lowest value in the dataset. This gives you a quick glance at the two historical extreme values. The formula to find the range is as follows:

$$\text{Range} = \max(x) - \min(x)$$

In Python, you can easily do this, as there are built-in functions that show the maximum and the minimum values given a set of data:

```
# Calculating the range
range_metric = max(cpi["CPIAUCSL"]) - min(cpi["CPIAUCSL"])
# Printing the result
print('The range of the dataset: ' + str(range_metric), '%')
```

The output of the preceding code should be as follows:

```
The range of the dataset: 16.5510 %
```

[Figure 3-3](#) shows the CPI values since 1951. The diagonal dashed line represents the range, and the horizontal dashed line represents the zero threshold.

The range of the CPI shows the size of the variations in inflation measures from one period to another. Yearly changes in inflation numbers vary from one country to another. Generally, developed countries such as France and the United States have stable variations (in times of stability), while emerging and frontier world countries such as Turkey and Argentina have more volatile and more extreme inflation numbers.

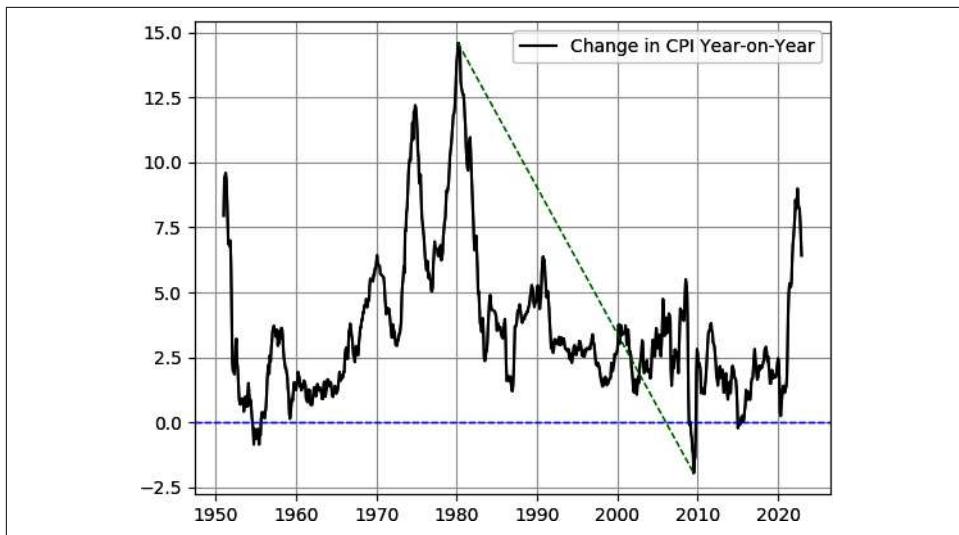


Figure 3-3. US CPI year-on-year change since 1951, with a diagonal dashed line that represents the range



The key takeaways from this section are as follows:

- The three key variability metrics that you should know are the variance, the standard deviation, and the range.
- The standard deviation is the square root of the variance. This is done so that it becomes comparable to the mean.
- The range is the difference between the highest and the lowest values in a dataset. It is a quick snapshot of the overall stretch of the observations.

Measures of Shape

Measures of shape describe the distribution of the values around the central tendency measures in a dataset. The mean and the standard deviation are the two factors that describe the normal distribution. The standard deviation depicts the spread or dispersion of the data, and the mean reflects the distribution's center.

A *probability distribution* is a mathematical function that describes the likelihood of different outcomes or events in a random experiment. In other words, it gives the probabilities of all possible values of a random variable.

There are many types of probability distributions, including discrete and continuous distributions. *Discrete distributions* take on a finite number of values. The best-known discrete distributions are the Bernoulli distribution, binomial distribution, and Poisson distribution.

Continuous distributions are used for random variables that can take on any value within a given range (such as stock prices and returns). The best-known distribution is the normal distribution.

The *normal distribution* (also known as the *Gaussian distribution*) is a type of continuous probability distribution that is symmetrical around the mean and has a bell shape. It is one of the most widely used distributions in statistical analysis and is often used to describe natural phenomena such as age, weight, and test scores. [Figure 3-4](#) shows the shape of a normal distribution.

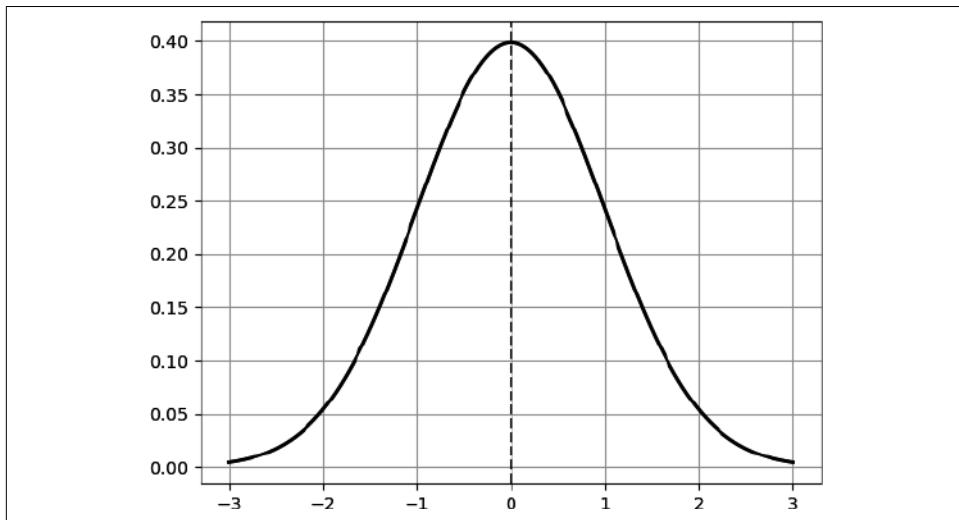


Figure 3-4. A normal distribution plot with mean = 0 and standard deviation = 1

You can generate [Figure 3-4](#) using the following code block:

```
# Importing libraries
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as stats
# Generate data for the plot
data = np.linspace(-3, 3, num = 1000)
# Define the mean and standard deviation of the normal distribution
mean = 0
std = 1
# Generate the function of the normal distribution
pdf = stats.norm.pdf(data, mean, std)
# Plot the normal distribution plot
```

```
plt.plot(data, pdf, '-.', color = 'black', lw = 2)
plt.axvline(mean, color = 'black', linestyle = '--')
plt.grid()
plt.show()
```



Since normally distributed variables are common, most statistical tests and models assume that the analyzed data is normal. With financial returns, they are assumed normal even though they experience a form of skew and kurtosis, two measures of shape discussed in this section.

In a normal distribution, the data is distributed symmetrically around the mean, which also means that the mean is equal to the median and to the mode. Furthermore, around 68% of the data falls within one standard deviation from the mean, around 95% falls within two standard deviations, and around 99.7% falls within three standard deviations. This property makes the normal distribution a useful tool for making inferences.

To sum up, what you should retain from the normal distribution is the following:

- The mean and the standard deviation describe the distribution.
- The mean splits the distribution halfway, making it equal to the median. Due to the symmetrical property, the mode is also equal to the mean and the median.

Now let's discuss the measures of shape. The first measure of shape is skewness. *Skewness* describes a distribution's asymmetry. It analyzes how far from being symmetrical the distribution deviates.

The skewness of a normal distribution is equal to zero. This means that the distribution is perfectly symmetrical around its mean, with an equal number of data points on either side of the mean.

A *positive skew* indicates that the distribution has a long tail to the right, which means that the mean is greater than the median because the mean is sensible to outliers, which will push it upward (therefore, to the right of the x -axis). Similarly, the mode will be the lowest value between the three central tendency measures. [Figure 3-5](#) shows a positive skew.

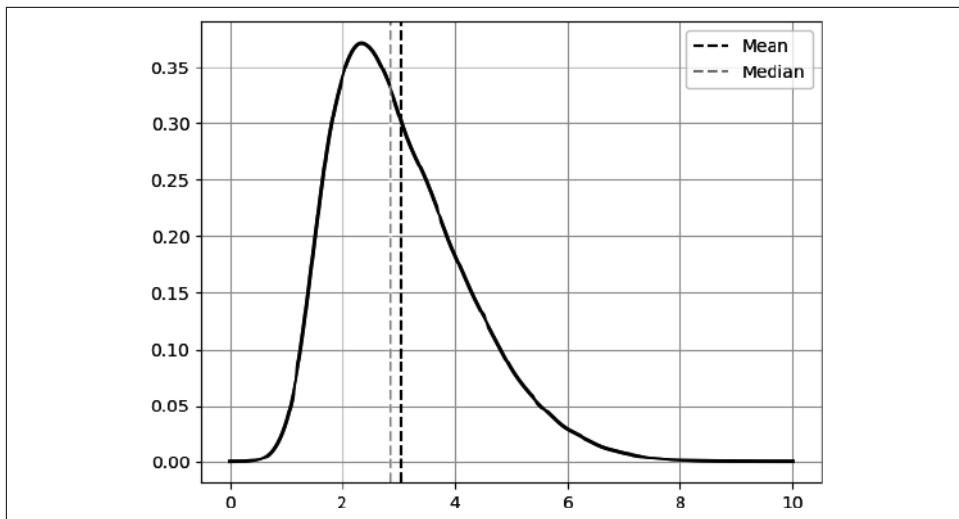


Figure 3-5. An example of a positively skewed distribution

A *negative skew* indicates that the distribution has a long tail to the left, which means that the mean is lower than the median. Similarly, the mode will be the greatest value between the three central tendency measures. [Figure 3-6](#) shows a negative skew.

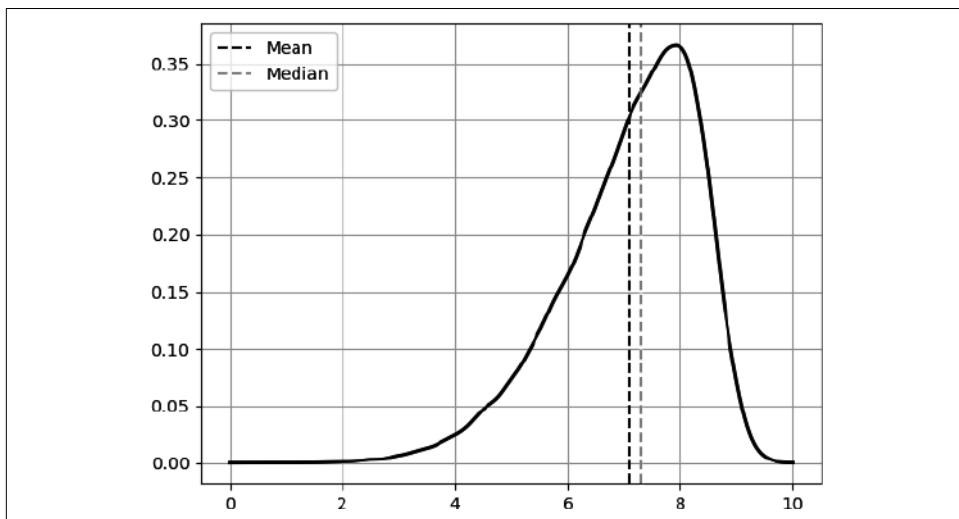


Figure 3-6. An example of a negatively skewed distribution



How can skewness be interpreted in the world of financial markets? If the distribution is positively skewed, it means that there are more returns above the mean than below it (the tail of the distribution is longer on the positive side).

If the distribution is negatively skewed, it means that there are more returns below the mean than above it (the tail of the distribution is longer on the negative side).

The skew of a returns series can provide information about the risk and return of an investment. For example, a positively skewed returns series may indicate that the investment has a potential for a few large gains with a risk of frequent small losses.

The formula to find skewness is as follows:

$$\tilde{\mu}_3 = \frac{\sum_{n=1}^i (x_i - \bar{x})^3}{N\sigma^3}$$

Let's check the skewness of the US CPI year-on-year data since 2003:

```
# Calculating the skew
skew = cpi_latest['CPIAUCSL'].skew()
# Printing the result
print('The skew of the dataset: ' + str(skew))
```

The output of the preceding code should be as follows:

The skew of the dataset: 1.17

The skew of the data is 1.17, but what does that mean? Let's chart the distribution of the data to facilitate interpretation. You can do this using the following code snippet:

```
# Plotting the histogram of the data
fig, ax = plt.subplots()
ax.hist(cpi['CPIAUCSL'], bins = 30, edgecolor = 'black', color = 'white')
# Add vertical lines for better interpretation
ax.axvline(mean, color='black', linestyle='--', label = 'Mean',
           linewidth = 2)
ax.axvline(median, color='grey', linestyle='-.', label = 'Median',
           linewidth = 2)
plt.grid()
plt.legend()
plt.show()
```

Figure 3-7 shows the result of the previous code snippet. The data is clearly positively skewed since the mean is greater than the median and the skewness is positive (above zero).

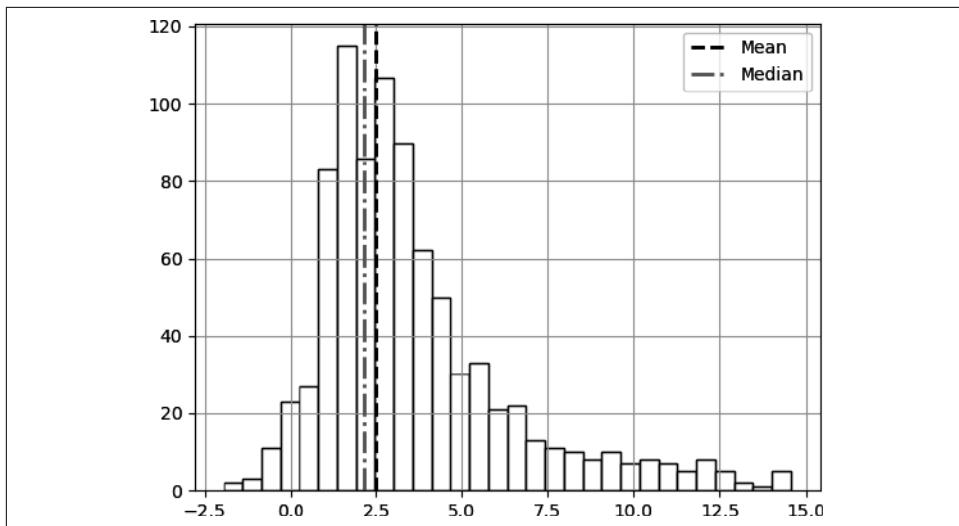


Figure 3-7. Data distribution of the US CPI year-on-year values, indicating positive skew

Remember, skewness is a measure of the asymmetry of a probability distribution. It therefore measures the degree to which the distribution deviates from normality. The rules of thumb to interpret skewness are as follows:

- If skewness is between -0.5 and 0.5 , the data is considered symmetrical.
- If skewness is between -1.0 and -0.5 or between 0.5 and 1.0 , the data is considered mildly skewed.
- If skewness is less than -1.0 or greater than 1.0 , the data is considered highly skewed.

What does a positive skew mean? In this case, 1.17 represents highly skewed data (in the positive side), which is in line with a monetary policy that favors inflation as the economy grows (with a few inflationary spikes that cause the skew).



It may be interesting to know that with a skewed distribution, the median may be the preferred metric since the mean tends to be pulled by outliers, thus distorting its value.

The next measure of shape is *kurtosis*, which is a description of the peakedness or flatness of a distribution relative to a normal distribution. Kurtosis describes the tails of a distribution, in particular, whether the tails are thicker or thinner than those of a normal distribution.

A normal distribution has a kurtosis of 3, which means it is a *mesokurtic* distribution. If a distribution has a kurtosis greater than 3, it is referred to as *leptokurtic*, meaning it has a higher peak and fatter tails than a normal distribution. If a distribution has a kurtosis less than 3, it is referred to as *platykurtic*, meaning it has a flatter peak and thinner tails than a normal distribution.

The formula to find kurtosis is as follows:

$$k = \frac{\sum_{n=1}^i (x_i - \bar{x})^4}{N\sigma^4}$$

Sometimes kurtosis is measured as *excess kurtosis* to give it a starting value of zero (for a normal distribution). This means that the kurtosis measure is subtracted from 3 so as to calculate the excess kurtosis. Let's calculate excess kurtosis for the US CPI year-on-year data:

```
# Calculating the excess kurtosis
excess_kurtosis = cpi_latest["CPIAUCSL"].kurtosis()
# Printing the result
print('The excess kurtosis of the dataset: ' + str(excess_kurtosis))
```

The output of the preceding code should be as follows:

The excess kurtosis of the dataset: 2.15

In the case of the US CPI year-on-year values of the past 20 years, excess kurtosis is 2.15, which is more in line with a leptokurtic (peakier with fatter tails) distribution. A positive value indicates a distribution more peaked than normal, and a negative kurtosis indicates a shape flatter than normal.

Understanding Inflation

Independent from statistics, it is interesting to know the terminology of what you are analyzing. *Inflation* is a decrease in purchasing power of the economic agents (such as households). A decrease in purchasing power means that agents can buy less over time with the same amount of money. This is also referred to as a general price increase. Inflation in the economic sense has the following forms:

Inflation

Controlled inflation is associated with steady economic growth and expansion. It is a desired attribute for a growing economy. Regulators monitor inflation and try to stabilize it to prevent social and economic distress.

Deflation

Whenever inflation is in the negative territory, it is referred to as deflation. Deflation is very dangerous for the economy, and as tempting as it may be for consumers who see a price decrease, deflation is a growth killer and may cause extended economic gluts that lead to unemployment and bearish stock markets.

Stagflation

This occurs when inflation is either high or rising while economic growth is slowing down. Simultaneously, unemployment remains high. It is one of the worst possible case scenarios.

Disinflation

This is a decrease in inflation but in the positive territory. For example, if this year's inflation is 2% while last year's inflation was 3%, you can say that there was disinflation on a yearly basis.

Hyperinflation

This is the nightmarish scenario that occurs when inflation goes out of control and experiences astronomical percent changes, such as a percentage change in the millions from year to year (as has famously occurred in Zimbabwe, Yugoslavia, and Greece).

The final metric we will discuss in the descriptive statistics category is quantiles. *Quantiles* are measures of both shape and variability since they provide information about the distribution of values (shape) and about the dispersion of those values (variability). The most used type of quantiles is the quartile.

Quartiles divide the dataset into four equal parts. This is done by arranging the data in order and then performing the split. Consider [Table 3-1](#) as an example.

Table 3-1. Numbers in ascending order

Value
1
2
4
5
7
8
9

The quartiles are as follows:

- The lower quartile (Q1) is the first quarter, which in this case is 2.
- The middle quartile (Q2) is also the median, which in this case is 5.
- The upper quartile (Q3) in this case is 8.

Mathematically, you can calculate Q1 and Q3 using the following formulas:

$$Q_1 = \left(\frac{n+1}{4} \right)$$

$$Q_3 = 3\left(\frac{n+1}{4} \right)$$

Keep in mind that the results of the formulas give you the ranking of the values but not the values themselves:

$$Q_1 = \left(\frac{7+1}{4} \right) = 2^{\text{nd}} \text{ term} = 2$$

$$Q_3 = 3\left(\frac{7+1}{4} \right) = 6^{\text{th}} \text{ term} = 8$$

The *interquartile range* (IQR) is the difference between Q3 and Q1 and provides a measure of the spread of the middle 50% of the values in a dataset. The IQR is robust to outliers (since it relies on middle values) and provides a brief summary of the spread of the bulk of the values. The IQR of the data in [Table 3-1](#) is 6 as per the following formula:

$$IQR = Q_3 - Q_1$$

$$IQR = 8 - 2 = 6$$

The IQR is a valuable indicator and can be used as an input or a risk metric in many different models. It can also be used to detect outliers in the data since it is immune to them. Also, the IQR can help evaluate the current volatility of the analyzed asset, which in turn can be used with other methods to create more powerful models. As understood, the IQR outperforms the range metric in terms of usefulness and interpretation as the former is prone to outliers.

Be careful when calculating quartiles, as there are many methods that use different calculations for the same dataset. Most importantly, make sure you use the same method throughout your analyses. The method used to calculate the quartiles in [Table 3-1](#) is called the *Tukey's hinges* method.



The key takeaways from this section are the following:

- The normal distribution is a continuous probability distribution that has a bell-shaped curve. The majority of the data clusters around the mean. The mean, median, and mode of a normal distribution curve are all equal.
- Skewness measures the asymmetry of a probability distribution.
- Kurtosis measures the peakedness of a probability distribution. Excess kurtosis is commonly used to describe the current probability distribution.
- Quantiles divide the arranged dataset into equal parts. The most commonly used quantiles are quartiles that divide the data into four equal parts.
- The IQR is the difference between the third quartile and the first quartile. It is immune to outliers and thus is very helpful in data analysis.

Visualizing Data

In Chapter 1, I presented the six steps in the data science process. Step 4 was data visualization. This section will show you a few ways to present data in a clear visual manner that allows you to interpret it.

Many types of statistical plots are commonly used to visualize data. Let's discuss some of them.

Scatterplots are used to graph the relationship between two variables through points that correspond to the intersection between the variables. To create a scatterplot, use the following code on the CPI data:

```
# Importing the required library
import matplotlib.pyplot as plt
# Resetting the index
cpi = cpi.reset_index()
# Creating the chart
fig, ax = plt.subplots()
ax.scatter(cpi['DATE'], cpi['CPIAUCSL'], color = 'black',
           s = 8, label = 'Change in CPI Year-on-Year')
plt.grid()
plt.legend()
plt.show()
```

Figure 3-8 shows the result of a scatterplot in time. This means that you have the CPI data as the first variable (*y*-axis) and time as the second variable (*x*-axis). However, scatterplots are more commonly used to compare variables; thus, removing the time variable can give more insights.

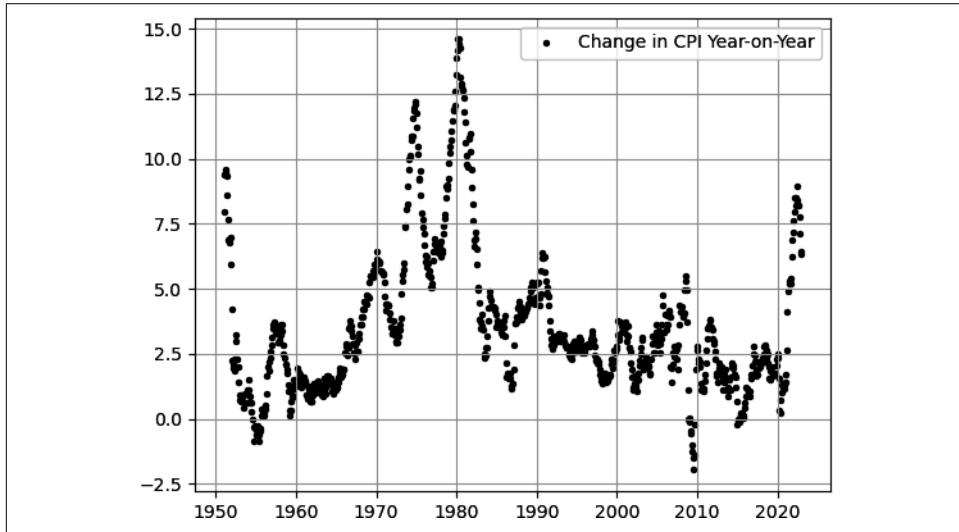


Figure 3-8. Scatterplot of US CPI data versus the time axis

Compare the UK CPI year-on-year change with the US CPI year-on-year change. Notice the positive association between the two in **Figure 3-9**, as higher values of one are correlated with higher values of the other. Correlation is a key measure that you will see in detail in the next section. The code to plot **Figure 3-9** is as follows:

```
# Setting the beginning and end of the historical data
start_date = '1995-01-01'
end_date   = '2022-12-01'
# Creating a dataframe and downloading the CPI data
cpi_us = pdr.DataReader('CPIAUCSL', 'fred', start_date, end_date)
cpi_uk = pdr.DataReader('GBRCPIALLMINMEI', 'fred', start_date, end_date)
# Dropping the NaN values from the rows
cpi_us = cpi_us.dropna()
cpi_uk = cpi_uk.dropna()
# Transforming the CPI into a year-on-year measure
cpi_us = cpi_us.pct_change(periods = 12, axis = 0) * 100
cpi_us = cpi_us.dropna()
cpi_uk = cpi_uk.pct_change(periods = 12, axis = 0) * 100
cpi_uk = cpi_uk.dropna()
# Creating the chart
fig, ax = plt.subplots()
ax.scatter(cpi_us['CPIAUCSL'], cpi_uk['GBRCPIALLMINMEI'],
           color = 'black', s = 8, label = 'Change in CPI Year-on-Year')
# Adding a few aesthetic elements to the chart
```

```

ax.set_xlabel('US CPI')
ax.set_ylabel('UK CPI')
ax.axvline(x = 0, color='black', linestyle = 'dashed', linewidth = 1)
ax.axhline(y = 0, color='black', linestyle = 'dashed', linewidth = 1)
ax.set_ylim(-2,)
plt.grid()
plt.legend()
plt.show()

```

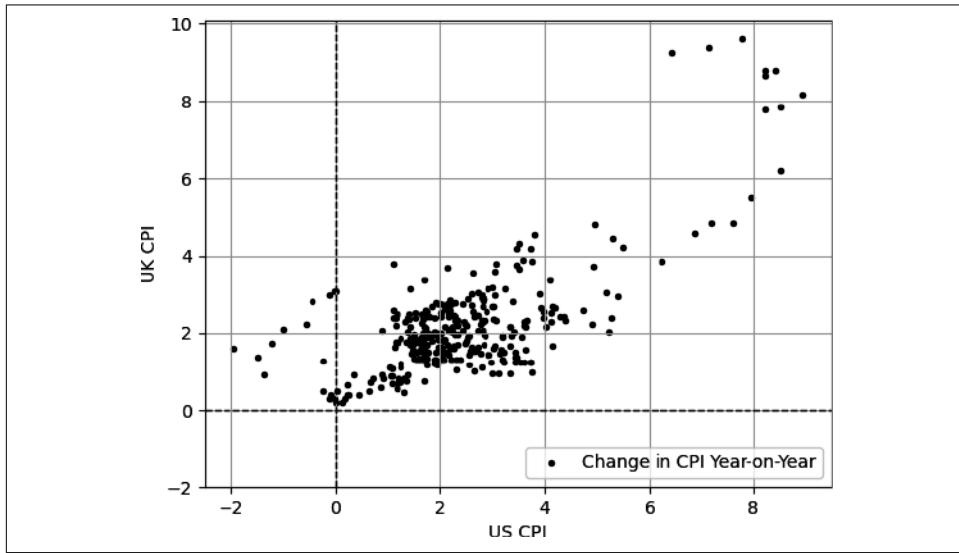


Figure 3-9. Scatterplot of UK CPI data versus US CPI data

Scatterplots are good when visualizing the correlation between data. They are also easy to draw and interpret. Generally, when the points are scattered in such a way that a diagonal upward-sloping line can be drawn to represent them, the correlation is assumed to be positive, since whenever variables on the x -axis increase, variables on the y -axis also increase.

On the other hand, when a diagonal downward-sloping line can be drawn to represent the different variables, a negative correlation may exist. A negative correlation implies that whenever variables on the x -axis move, it is likely that variables on the y -axis move in the opposite way.

[Figure 3-10](#) shows a best-fit line between the two inflation datasets from [Figure 3-9](#). Notice how it is upward sloping.

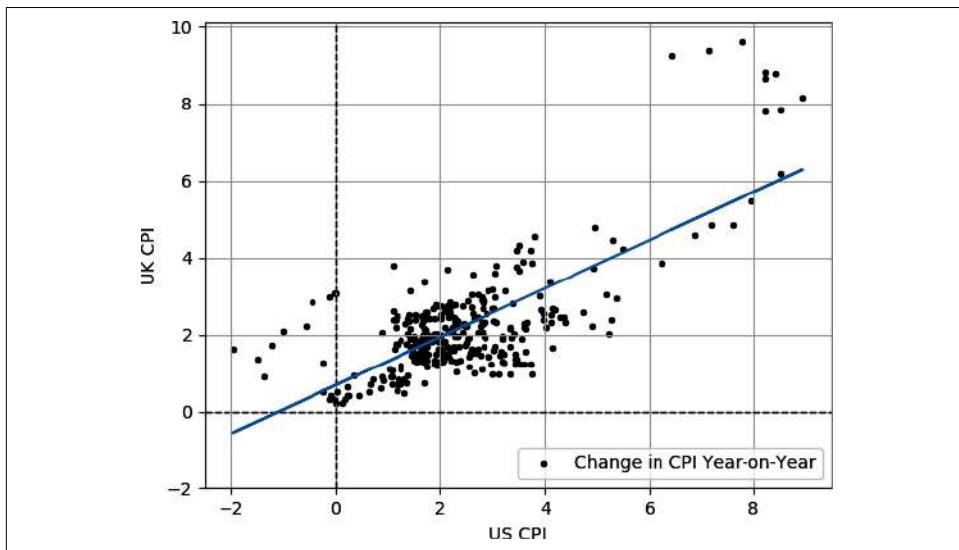


Figure 3-10. Scatterplot of UK CPI data versus US CPI data with a best-fit line

Let's now move to another charting method. *Line plots*, the most basic type of plot, are essentially scatterplots that are joined and are mostly charted against the time axis (*x*-axis). You saw line plots in Figures 3-1 and 3-2.

The advantage of line plots is their simplicity and ease of implementation. They also show the evolution of the series through time, which helps in detecting trends and patterns. In [Chapter 5](#), you will learn about candlestick plots, a more elaborate way to plot financial time series. [Figure 3-11](#) shows a basic line plot of US CPI data since 1951.

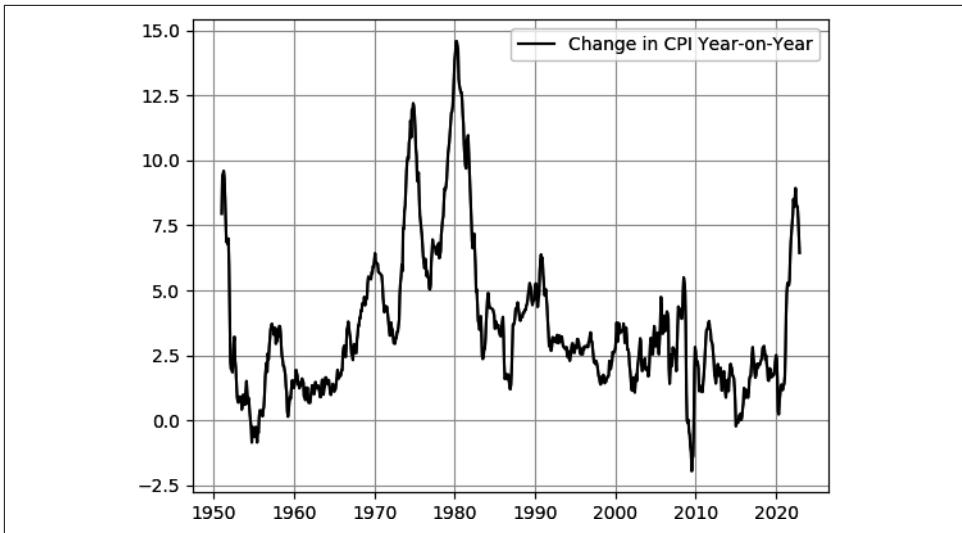


Figure 3-11. Line plot of US CPI data versus the time axis

To create Figure 3-11, you can use the following code snippet:

```
# Creating the chart
plt.plot(cpi['DATE'], cpi['CPIAUCSL'], color = 'black',
         label = 'Change in CPI Year-on-Year')
plt.grid()
plt.legend()
plt.show()
```

Next up are *bar plots*, which display the distribution of variables (generally, categorical). Figure 3-12 shows a bar plot of the US CPI data since the beginning of 2022.

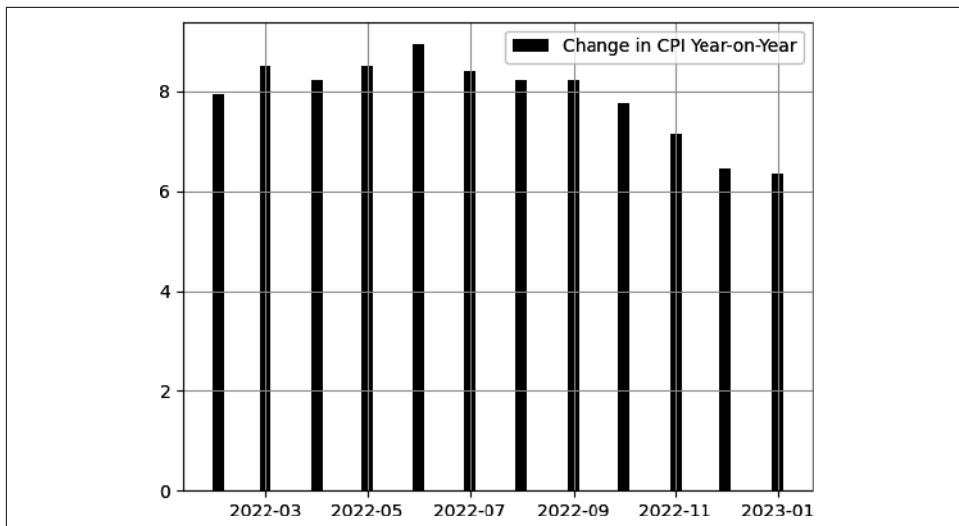


Figure 3-12. Bar plot of US CPI data versus the time axis

To create Figure 3-12, you can use the following code snippet:

```
# Taking the values of the previous twelve months
cpi_one_year = cpi.iloc[-12:]
# Creating the chart
plt.bar(cpi_one_year['DATE'], cpi_one_year['CPIAUCSL'],
        color = 'black', label = 'Change in CPI Year-on-Year', width = 7)
plt.grid()
plt.legend()
plt.show()
```

Bar plots can be limited for plotting continuous data such as the US CPI or stock prices. They can also be misleading when the scale is off. Bar plots are also not recommended for large datasets since they clutter up the space. For the latter reason, histograms are a better fit.

A *histogram* is a specific sort of bar chart that is used to display the frequency distribution of continuous data by using bars to represent statistical information. It indicates the number of observations that fall into the class or bin of values. An example of a histogram is shown in Figure 3-13 (also see Figure 3-7).

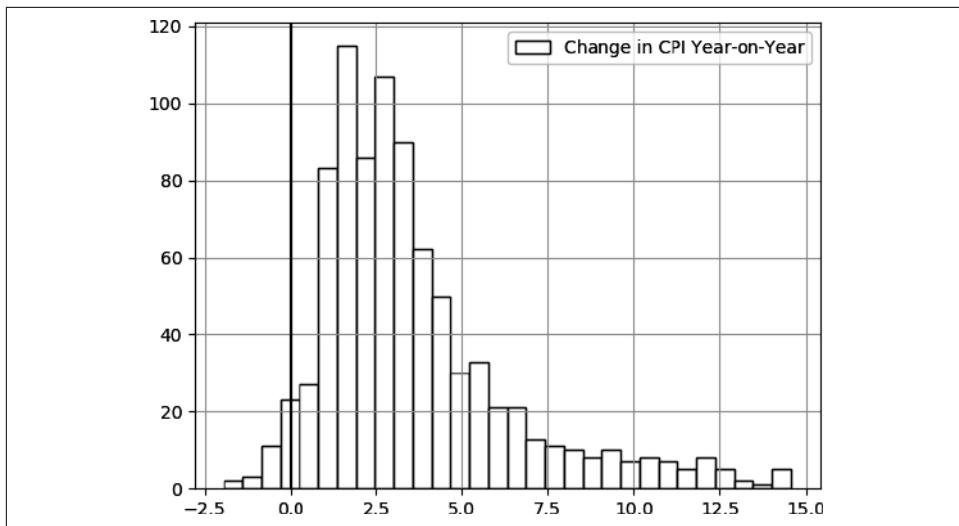


Figure 3-13. Histogram plot of US CPI data

To create Figure 3-13, you can use the following code snippet:

```
# Creating the chart
fig, ax = plt.subplots()
ax.hist(cpi['CPIAUCSL'], bins = 30, edgecolor = 'black',
        color = 'white', label = 'Change in CPI Year-on-Year')
# Add vertical lines for better interpretation
ax.axvline(0, color = 'black')
plt.grid()
plt.legend()
plt.show()
```

Notice how the bar plot is charted against the time axis, but the histogram plot does not have a time horizon because it is a group of values with the aim of showing the overall distribution points. Visually, you can see the positive skewness of the distribution.

Another classic plotting technique in statistics is the *box and whisker plot*. It is used to visualize the distribution of continuous variables while including the median and the quartiles, as well as the outliers. The way to understand the box and whisker plot is as follows:

- The box represents the IQR. The box is drawn between the first quartile and the third quartile. The height of the box indicates the spread of the data in this range.
- The line inside the box represents the median.

- The whiskers extend from the top and bottom of the box to the highest and lowest data points that are still within 1.5 times the IQR. These data points are called *outliers* and are represented as individual points on the plot.

Figure 3-14 shows a box and whisker plot on the US CPI data since 1950.

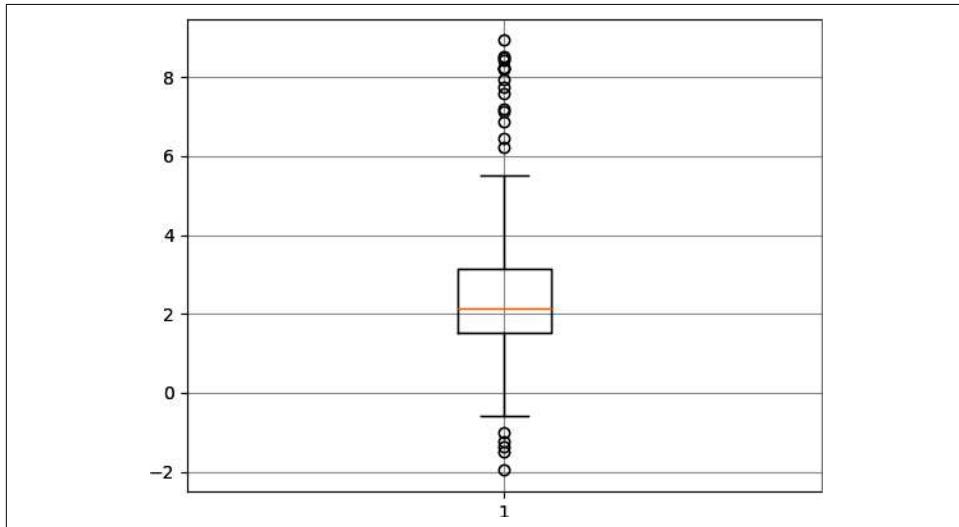


Figure 3-14. Box and whisker plot of US CPI data

You can also plot the data without the outliers (any value that lies more than 1.5 times the length of the box from either end of the box). To create Figure 3-14, you can use the following code snippet:

```
# Creating the chart
cpi_latest = cpi.iloc[-240:]
fig, ax = plt.subplots()
ax.boxplot(cpi_latest['CPIAUCSL'])
plt.grid()
plt.legend()
plt.show()
```

To remove the outliers from the plot, you simply use the following tweak:

```
# Replace the corresponding code line with the following
fig, ax = plt.subplots()
ax.boxplot(cpi_latest['CPIAUCSL'], showfliers = False)
```

This will give you Figure 3-15.

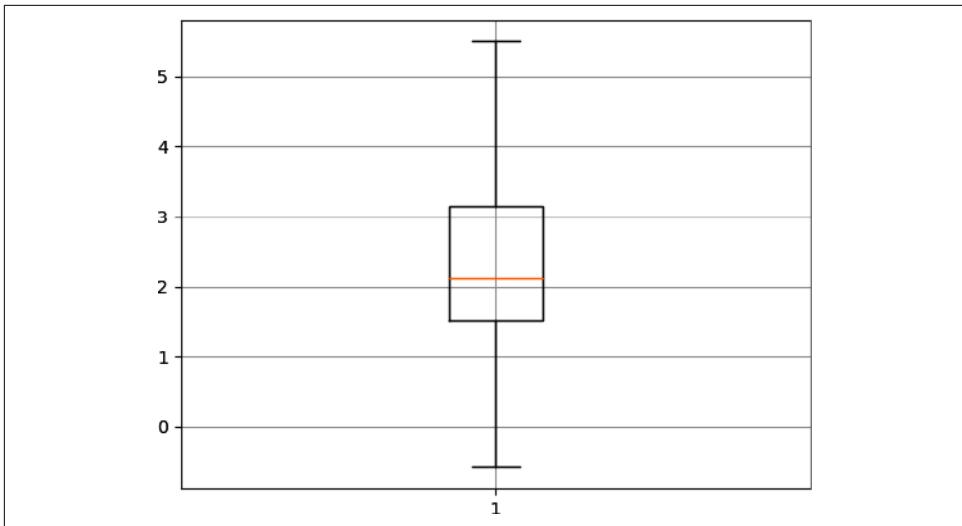


Figure 3-15. Box and whisker plot of US CPI data with no outliers

Many more data visualization techniques exist, such as *heatmaps* (commonly used with correlation data and temperature mapping) and *pie charts* (commonly used for budgeting and segmentation). Which technique to use depends on what you need to understand and what fits better with your needs. For example, a line plot is better suited for time series that only have one feature (e.g., only the close price of a certain security is available). A histogram plot is better suited for use with probability distribution data.



The key takeaways from this section are the following:

- Which data visualization technique to use depends on the type of analysis and interpretation you want to perform. Some plots are better suited for use with certain types of data.
- Data visualization helps with the initial interpretation before confirming it numerically.
- You are more likely to use line plots and candlestick plots when dealing with financial time series.

Correlation

Correlation is a measure used to calculate the degree of the linear relationship between two variables. It is a number between -1.0 and 1.0 , with -1.0 designating a strong negative relationship between the variables and 1.0 designating a strong positive relationship.

A value of zero indicates that there is no linear association between the variables. However, correlation does not imply causation. Two variables are said to be correlated if they move in the same direction, but this does not imply that one causes the other to move or that they move as a result of the same events.

Most people agree that some assets have natural correlations. For instance, because they are both part of the same industry and are affected by the same trends and events, the stocks of Apple and Microsoft are positively connected (which means their general trend is in the same direction). [Figure 3-16](#) shows the chart between the two stocks. Notice how they move together.



Figure 3-16. Apple and Microsoft stock prices since 2021

The tops and bottoms of both stocks occur at almost the exact same time. Similarly, as the United States and the UK have similar economic drivers and impacts, they are also likely to have positively correlated inflation numbers, as you saw earlier in the chapter.

Checking for correlation is done through visual interpretation and mathematical formulas. Before seeing an example, let's discuss the roots of calculating correlation so that you know where it comes from and what its limitations are.



Simply put, to calculate correlation, you need to measure how close the points in a scatterplot of the two variables are to a straight line. The more they look like a straight line, the more they are positively correlated, hence the term *linear correlation*.

There are two main methods for calculating correlation: the Spearman method and the Pearson method.¹

The *Pearson* correlation coefficient is a measure of the linear association between two variables calculated from the standard deviation and the covariance between them.

Covariance calculates the average of the difference between the means of the two variables. If the two variables have a tendency to move together, the covariance is positive, and if the two variables typically move in opposite directions, the covariance is negative. It ranges between positive infinity and negative infinity.

The formula for calculating the covariance between variables x and y is as follows:

$$cov_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n}$$

Therefore, covariance is the sum of the products of the average deviations between the variables and their respective means (i.e., covariance measures the degree of their association). An average is taken to normalize this calculation. The Pearson correlation coefficient is calculated as follows:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

Simplifying the previous correlation formula gives you the following:

$$r_{xy} = \frac{cov_{xy}}{\sigma_x \sigma_y}$$

Therefore, the Pearson correlation coefficient is simply the covariance between two variables divided by the product of their standard deviation. Let's calculate the correlation between the US CPI year-on-year values and the UK CPI year-on-year values. The intuition is that the correlation is above zero, as the UK and the US are

¹ Among others, but these two ways are the most popular representations.

economically related. The following code block calculates the Pearson correlation coefficient for the two time series:

```
# Importing the required libraries
import pandas_datareader as pdr
import pandas as pd
# Setting the beginning and end of the historical data
start_date = '1995-01-01'
end_date = '2022-12-01'
# Creating a dataframe and downloading the CPI data
cpi_us = pdr.DataReader('CPIAUCSL', 'fred', start_date, end_date)
cpi_uk = pdr.DataReader('GBRCPIALLMINMEI', 'fred', start_date, end_date)
# Dropping the nan values from the rows
cpi_us = cpi_us.dropna()
cpi_uk = cpi_uk.dropna()
# Transforming the US CPI into a year-on-year measure
cpi_us = cpi_us.pct_change(periods = 12, axis = 0) * 100
cpi_us = cpi_us.dropna()
# Transforming the UK CPI into a year-on-year measure
cpi_uk = cpi_uk.pct_change(periods = 12, axis = 0) * 100
cpi_uk = cpi_uk.dropna()
# Joining both CPI data into one dataframe
combined_cpi_data = pd.concat([cpi_us['CPIAUCSL'],
                                cpi_uk['GBRCPIALLMINMEI']], axis = 1)
# Calculating Pearson correlation
combined_cpi_data.corr(method = 'pearson')
```

The output is as follows:

	CPIAUCSL	GBRCPIALLMINMEI
CPIAUCSL	1.000000	0.732164
GBRCPIALLMINMEI	0.732164	1.000000

The correlation between the two is a whopping 0.73. This is in line with the expectations. Pearson correlation is usually used with variables that have proportional changes and are normally distributed.

Spearman's rank correlation is a nonparametric rank correlation that measures the strength of the relationship between the variables. It is suitable for variables that do not follow a normal distribution.



Remember, financial returns are not normally distributed but are sometimes treated that way for simplicity.

Unlike Pearson correlation, Spearman's rank correlation takes into account the order of the values, rather than the actual values. To calculate Spearman's rank correlation, follow these steps:

- Rank the values of each variable. This is done by inputting 1 instead of the smallest variable and inputting the length of the dataset instead of the largest number.
- Calculate the difference in ranks. Mathematically, the difference in ranks is represented by the letter d in the mathematical formula to come. Then, calculate their squared differences.
- Sum the squared differences you have calculated from step 2.
- Use the following formula to calculate Spearman's rank correlation:

$$\rho = 1 - \frac{6\sum_{i=1}^n d_i^2}{n^3 - n}$$

As with Pearson correlation, Spearman's rank correlation also ranges from -1.00 to 1.00 with the same interpretation.



Strong positive correlations are generally upward of 0.70, while strong negative correlations are generally downward of -0.70.

The following code block calculates Spearman's rank correlation coefficient for the two time series:

```
# Calculating Spearman's rank correlation
combined_cpi_data.corr(method = 'spearman')
```

The output is as follows:

	CPIAUCSL	GBRCPIALLMINMEI
CPIAUCSL	1.000000	0.472526
GBRCPIALLMINMEI	0.472526	1.000000

Let's answer a very important question after getting this difference in results. Why are the two measures so different?

The first thing to keep in mind is what they measure. Pearson correlation measures the linear relationship (trend) between the variables while Spearman's rank correlation measures the monotonic trend. The word *monotonic* refers to moving in the same direction but not exactly at the same rate or magnitude. Also, Spearman's rank correlation transforms the data to an ordinal type (through the ranks) as opposed to Pearson correlation, which uses the actual values.

Autocorrelation (also referred to as *serial correlation*) is a statistical method used to look at the relationship between a given time series and a lagged version of it. It is

generally used to predict future values through patterns in data, such as seasonality or trends. Autocorrelation is therefore the values' relationship with the previous values—for example, comparing each day's Microsoft stock price to the preceding day and seeing if there is a discernible correlation there. Algorithmically speaking, this can be represented as shown in [Table 3-2](#).

Table 3-2. Lagged values

	t		t-1
\$	1.25	\$	1.65
\$	1.77	\$	1.25
\$	1.78	\$	1.77
\$	1.25	\$	1.78
\$	1.90	\$	1.25

Each row represents a time period. Column t is the current price and column $t-1$ is the previous price put on the row that represents the present. This is done when creating machine learning models to understand the relationship between the current values and the previous ones at every time step (row).

Positive autocorrelation frequently occurs in trending assets and is associated with the idea of persistence (trend following). Negative autocorrelation is exhibited in ranging markets and is associated with the idea of antipersistence (mean reversion).



Measures of short-term correlation between different time series (e.g., NVIDIA and Oracle stocks) are typically computed using returns (such as differences) on prices rather than real prices. However, it is possible to utilize the prices directly to identify long-term trends.

The following code block calculates the autocorrelation of the US CPI year-on-year values:

```
# Creating a dataframe and downloading the CPI data
cpi = pdr.DataReader('CPIAUCSL', 'fred', start_date, end_date)
# Transforming the US CPI into a year-on-year measure
cpi = cpi.pct_change(periods = 12, axis = 0) * 100
cpi = cpi.dropna()
# Transforming the data frame to a series structure
cpi = cpi.iloc[:,0]
# Calculating autocorrelation with a lag of 1
print('Correlation with a lag of 1 = ', round(cpi.autocorr(lag = 1), 2))
# Calculating autocorrelation with a lag of 6
print('Correlation with a lag of 6 = ', round(cpi.autocorr(lag = 6), 2))
# Calculating autocorrelation with a lag of 12
print('Correlation with a lag of 12 = ', round(cpi.autocorr(lag = 12), 2))
```

A lag of 12 means that every data value is compared to the data from 12 periods ago and then a correlation measure is calculated. The output of the code is as follows:

```
Correlation with a lag of 1 =  0.97
Correlation with a lag of 6 =  0.65
Correlation with a lag of 12 =  0.17
```

Now, before proceeding to the next section, let's revert back to information theory and discuss one interesting correlation coefficient that is able to pick up on nonlinear relationships.

The *maximal information coefficient* (MIC) is a nonparametric measure of association between two variables designed to handle large and complex data. It is generally seen as a more robust alternative to traditional measures of correlation, such as Pearson correlation and Spearman's rank correlation. Introduced by David N. Reshef et al.,² the MIC uses concepts from information theory that you saw in [Chapter 2](#).

The MIC measures the strength of the association between two variables by counting the number of cells in a contingency table that are maximally informative about the relationship between the variables. The MIC value ranges from 0 to 1, with higher values indicating stronger associations. It can handle high-dimensional data and can identify nonlinear relationships between variables.

It is, however, nondirectional, which means that values close to 1 only suggest a strong correlation between the two variables; they do not say whether the correlation is positive or negative. In other words, the mutual information between the two variables within each bin is calculated after the range of each variable has been divided into a set of bins. The strength of the association between the two variables is then estimated using the maximum mutual information value across all bins.

Let's check out a practical example that showcases the strength of the MIC in detecting nonlinear relationships. The following example simulates two wave series (sine and cosine). Intuitively, looking at [Figure 3-17](#), it seems that there is a lag-lead relationship between the two.

² David N. Reshef et al., "Detecting Novel Associations in Large Data Sets," *Science* 334, no. 6062 (December 2011): 1518-24.

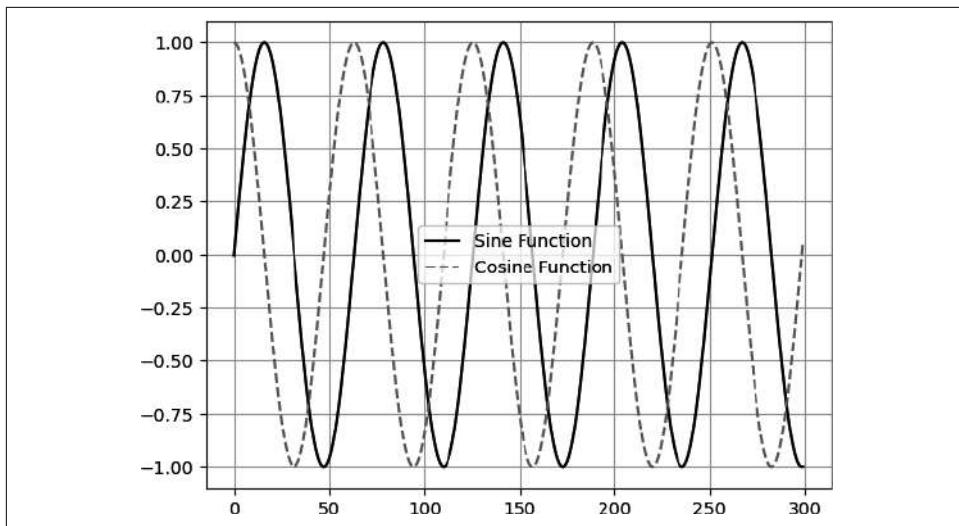


Figure 3-17. The two wave series showing a form of nonlinear relationship

The following Python code snippet creates the two time series and plots Figure 3-17:

```
# Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
# Setting the range of the data
data_range = np.arange(0, 30, 0.1)
# Creating the sine and the cosine waves
sine = np.sin(data_range)
cosine = np.cos(data_range)
# Plotting
plt.plot(sine, color = 'black', label = 'Sine Function')
plt.plot(cosine, color = 'grey', linestyle = 'dashed',
         label = 'Cosine Function')
plt.grid()
plt.legend()
```

Now the job is to calculate the three correlation measures and analyze their results. For simplicity, the code will be omitted from this section and will be shared in [Chapter 6](#), as there are a few things to understand in Python first. The results are as follows:

```
Correlation | Pearson:  0.035
Correlation | Spearman:  0.027
Correlation | MIC:  0.602
```

Let's interpret the results:

- *Pearson correlation*: Notice the absence of any type of correlation here due to it missing out on the nonlinear association.
- *Spearman's rank correlation*: The same situation applies here with an extremely weak correlation.
- *MIC*: The measure returned a strong relationship of 0.60 between the two, which is closer to reality. It seems that the MIC states that both waves have a strong, albeit nonlinear, relationship.

The MIC is useful in economic analysis, financial analysis, and even finding trading signals if used properly. Nonlinear relationships are abundant in such complex fields, and being able to detect them may provide a sizable edge.



The key takeaways from this section are the following:

- Correlation is a measure that is used to calculate the degree of the linear relationship between variables. It is a number between -1.0 and 1.0, with -1.0 designating a strong negative relationship between the variables and 1.0 designating a strong positive relationship.
- There are two main types of correlation: Spearman's rank correlation and Pearson correlation. Both have their advantages and limitations.
- Autocorrelation is the correlation of the variable with its own lagged values. For example, if the autocorrelation of Walmart's stock returns is positive, it denotes a trending configuration.
- Correlation measures can also refer to nonlinear relationships when you use the right tool, for example, the MIC.

The Concept of Stationarity

Stationarity is a key concept in statistical analysis and machine learning. *Stationarity* occurs when the statistical characteristics of the time series (mean, variance, etc.) are constant over time. In other words, no discernable trend is detectable when plotting the data across time.

The different learning models rely on data stationarity, as it is one of the basics of statistical modeling, and this is mainly for simplicity. In finance, price time series are not stationary, as they show trends with varying variance (volatility). Take a look at [Figure 3-18](#) and see if you can detect a trend. Would you say that this time series is stationary?

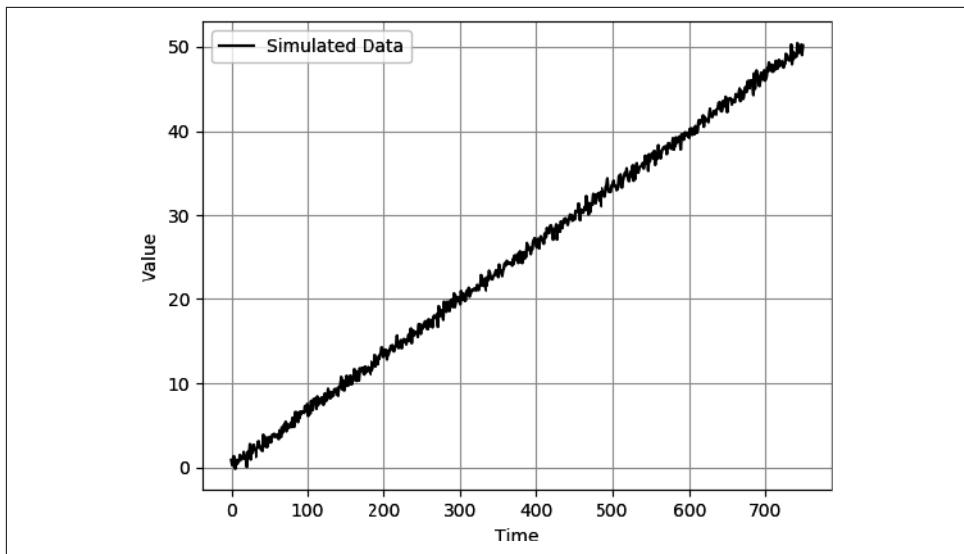


Figure 3-18. Simulated data with a varying mean across time

Naturally, the answer is no, as a rising trend is clearly in progress. States like this are undesirable for statistical analyses and machine learning. Luckily, there are transformations that you can apply to the time series to make them stationary. But first, let's see how to check for stationarity the mathematical way, as the visual way does not prove anything. The right way to deal with data stationarity problems is to follow these steps:

1. Check for stationarity using the different statistical tests that you will see in this section.
2. If the tests show data stationarity, you are ready to use the data for the algorithms. If the tests show that the data is not stationary, you have to proceed to step 3.
3. Subtract every value from the value before it (difference the values).
4. Recheck for stationarity using the same tests on the new transformed data.
5. If the test shows data stationarity, then you have successfully transformed your data. Otherwise, redo the transformation and check again until you have stationary data.



Ascending or descending time series have varying mean and variances through time and are therefore most likely nonstationary. There are exceptions to this, and you will see why later.

Remember, the aim of stationarity is stable and constant mean and variance over time. Therefore, when you look at [Figure 3-19](#), what can you infer?

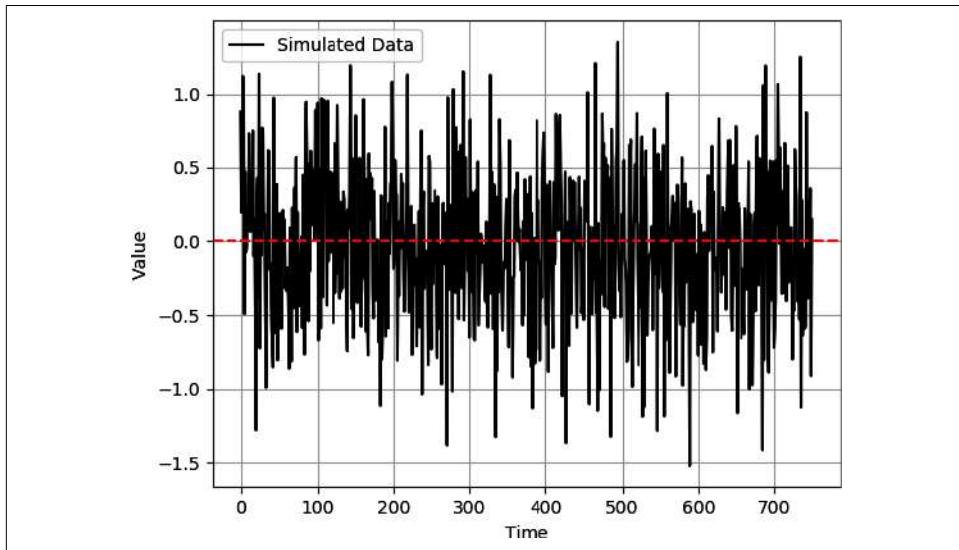


Figure 3-19. Simulated data with a mean around zero across time

Visually, it looks like the data does not have a trend, and it also looks like it fluctuates around a stable mean with stable variance around it. The first impression is that the data is stationary. Of course, this must be proved through statistical tests.

The first and most basic test is the *augmented Dickey—Fuller* (ADF) test. This tests for stationarity using hypothesis testing.

The ADF test searches for a unit root in the data. A *unit root* is a property of nonstationary data, and in the context of time series analysis, it refers to a characteristic of a stochastic process where the series has a root equal to 1. In simpler terms, it means that its statistical properties, such as the mean and variance, change over time. Here's what you need to know:

- The null hypothesis assumes the presence of a unit root. This means that if you are trying to prove that the data is stationary, you are looking to reject the null hypothesis (as seen in [“Sampling and Hypothesis Testing” on page 25](#)).
- The alternative hypothesis is therefore the absence of a unit root and the stationarity of the data.
- The p-value obtained from the test must be less than the significance level chosen (in most cases, it is 5%).

Let's test the US CPI year-on-year data for stationarity. The following code snippet checks for stationarity using the ADF test:

```
# Importing the required library
from statsmodels.tsa.stattools import adfuller
# Applying the ADF test on the CPI data
print('p-value: %f' % adfuller(cpi)[1])
```

The output of the code is as follows:

```
p-value: 0.0152
```

Assuming a 5% significance level, it seems that it is possible to accept that the year-on-year data is stationary (however, if you want to be stricter and use a 1% significance level, then the p-value suggests that the data is nonstationary). Either way, even looking at the chart can make you scratch your head. Remember that in [Figure 3-11](#), the yearly changes in the US CPI seem to be stable but do not resemble stationary data. This is why numerical and statistical tests are used.

Now, let's apply the code on the raw US CPI data and not take the year-on-year changes. Here's the code:

```
# Creating a dataframe and downloading the CPI data
cpi = pd.read_csv('CPIAUCSL', 'fred', start_date, end_date)
# Applying the ADF test on the CPI data
print('p-value: %f' % adfuller(cpi)[1])
```

The output of the code is as follows:

```
p-value: 0.999
```

Clearly, the p-value is greater than all the significance levels, which means that the time series is nonstationary. Let's sum up these results:

- It seems that you can reject the null hypothesis using a 5% significance level when it comes to the year-on-year changes in the US CPI data. The dataset is assumed to be stationary.
- It seems that you cannot reject the null hypothesis using a 5% significance level when it comes to the raw values of the US CPI data. The dataset is assumed to be nonstationary.

This becomes obvious when you plot the raw values of the US CPI data, as shown in [Figure 3-20](#).

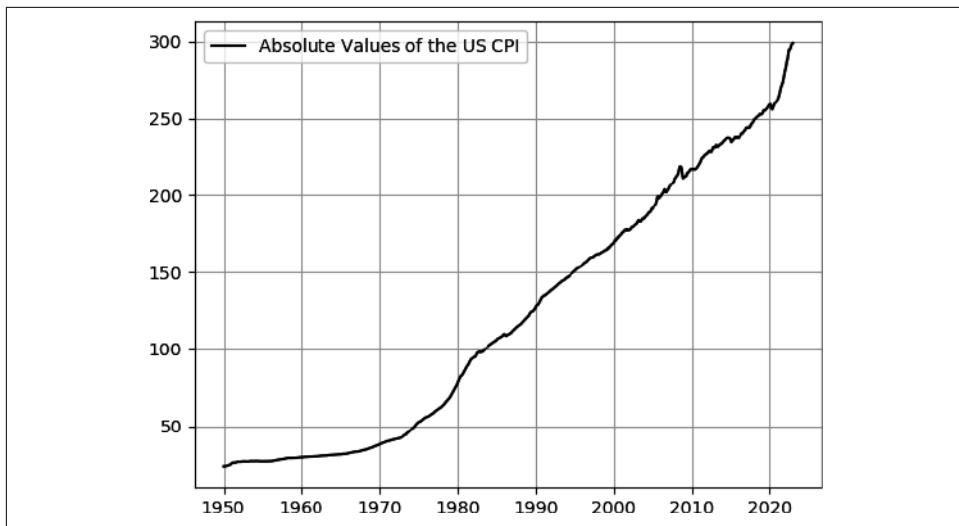


Figure 3-20. Absolute values of the US CPI data showing a clearly trending nature

The other test that you must be aware of is the *Kwiatkowski–Phillips–Schmidt–Shin* (KPSS) test, which is also a statistical test with the aim of determining whether the time series is stationary or nonstationary. However, the KPSS test can detect stationarity in trending time series, which makes it a powerful tool.

Trending time series can actually be stationary on the condition they have a stable mean.



The ADF test has a null hypothesis that argues for nonstationarity and an alternative hypothesis that argues for stationarity. The KPSS test has a null hypothesis that argues for stationarity and an alternative hypothesis that argues for nonstationarity.

Before analyzing the inflation data, let's see how a trending time series can be stationary. Remember that stationarity refers to a stable mean and standard deviation, so if somehow you have a gradually ascending or descending time series with stable statistical properties, it may be stationary. The next code snippet simulates a sine wave and then adds a touch of trending to it:

```
# Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
# Creating the first time series using sine waves
length = np.pi * 2 * 5
sinewave = np.sin(np.arange(0, length, length / 1000))
# Creating the second time series using trending sine waves
sinewaveAscending = np.sin(np.arange(0, length, length / 1000))
```

```

# Defining the trend variable
a = 0.01
# Looping to add a trend factor
for i in range(len(sinewaveAscending)):
    sinewaveAscending[i] = a + sinewaveAscending[i]
    a = 0.01 + a

```

Plotting the two series, as shown in [Figure 3-21](#), shows that the trending sine wave seems to be stable. But let's prove this through statistical tests.

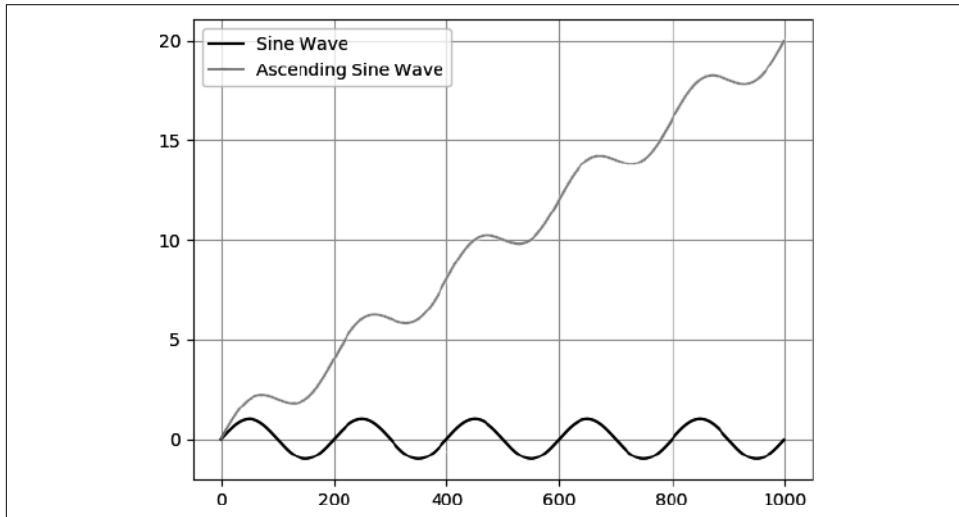


Figure 3-21. A normal sine wave simulated series with a trending sine wave

[Figure 3-21](#) is generated using the following code (make sure you have defined the series using the previous code block):

```

# Plotting the series
plt.plot(sinewave, label = 'Sine Wave', color = 'black')
plt.plot(sinewaveAscending, label = 'Ascending Sine Wave',
         color = 'grey')
plt.grid()
plt.legend()
plt.show()

```

Let's try the ADF test on both series and see what the results are:

```

# ADF testing | Normal sine wave
print('p-value: %f' % adfuller(sinewave)[1])
# ADF testing | Ascending sine wave
print('p-value: %f' % adfuller(sinewaveAscending)[1])

```

The output is as follows:

```

p-value: 0.000000 # For the sine wave
p-value: 0.989341 # For the ascending sine wave

```

Clearly, the ADF test is consistent with the idea that trending markets cannot be stationary. But what about the KPSS test? The following code uses the KPSS test on the same data to check for stationarity:

```
# Importing the KPSS library
from statsmodels.tsa.stattools import kpss
# KPSS testing / Normal sine wave
print('p-value: %f' % kpss(sinewave)[1])
# KPSS testing / Ascending sine wave
print('p-value: %f' % kpss(sinewaveAscending)[1])
# KPSS testing while taking into account the trend / Ascending sine wave
print('p-value: %f' % kpss(sinewaveAscending, regression = 'ct')[1])
'''

The 'ct' argument is used to check if the dataset is stationary
around a trend. By default, the argument is 'c' which is used
to check if the data is stationary around a constant.
'''
```

The output is as follows:

```
p-value: 0.10 # For the sine wave
p-value: 0.01 # For the ascending sine wave without trend consideration
p-value: 0.10 # For the ascending sine wave with trend consideration
```

Remember that the null hypothesis of the KPSS test is that the data is stationary; therefore, if the p-value is greater than the significance level, the data is considered stationary since it is not possible to reject the null hypothesis.

The KPSS statistic, when taking into account the trend, states that the ascending sine wave is a stationary time series. This is a basic example of how you can find stationary data in trending time series.

Let's test the US CPI year-on-year data for stationarity. The following code snippet checks for stationarity using the KPSS test:

```
# Applying the KPSS (no trend consideration) test on the CPI data
print('p-value: %f' % kpss(cpi)[1])
# Applying the KPSS (with trend consideration) test on the CPI data
print('p-value: %f' % kpss(cpi, regression = 'ct')[1])
```

The output of the code is as follows:

```
p-value: 0.010000 # without trend consideration
p-value: 0.010000 # with trend consideration
```

It seems that the results from the KPSS test contradict the results from the ADF test. This may happen from time to time, and differencing may solve the issue (bear in mind that the year-on-year data is already a differenced time series from the absolute CPI values, but some time series may need more than one differencing to become stationary, and it also depends on the period of differencing). The safest solution in this case is to transform the data again.

Before finishing this section on stationarity, let's discuss a complex topic that you will see in action in [Chapter 9](#). Transforming the data may cause the unusual problem of memory loss. In his book *Advances in Financial Machine Learning* (Wiley), Marcos López de Prado proposed a technique called fractional differentiation with the aim of making data stationary while preserving some memory.

When a nonstationary time series is differenced to make it stationary, memory loss occurs, which is another way of saying that the autocorrelation between the values is greatly reduced, thus removing the trend component and the DNA of the underlying asset. The degree of differencing and the persistence of the autocorrelation structure in the original series determines how much memory loss occurs.



The key takeaways from this section are the following:

- Stationarity refers to the concept of stable mean and variance through time. It is a desired characteristic, as most machine learning models rely on it.
- Financial price time series are most likely nonstationary and require first-order differencing to become stationary and ready for statistical modeling. Some may even require a second order transformation to become stationary.
- The ADF and KPSS tests check for stationarity in the data, with the latter being able to check for stationarity in trending data.
- Trending data may be stationary. Although this characteristic is rare, the KPSS test can detect the stationarity, while the ADF test cannot.

Regression Analysis and Statistical Inference

Whereas descriptive statistics describe data and extract as much information as possible from it, inferential statistics uses the data or a sample of the data to make inferences (forecasts). The main tool in statistical inference is linear regression.

Linear regression is a basic machine learning algorithm you will see in this book in [Chapter 7](#), when we cover the other machine learning algorithms. Hence, we will only briefly discuss regression analysis in this section. The most basic form of a linear regression equation is as follows:

$$y = \alpha + \beta x + \epsilon$$

y is the dependent variable; it is what you want to forecast

x is the independent variable; it is what you use as an input to forecast y

α is the expected value of the dependent variable when the independent variables are equal to zero

β represents the change in the dependent variable per unit change in the independent variable

ϵ is the residual or the unexplained variation

The basic linear regression equation states that a dependent variable (what you want to forecast) is explained by a constant, a sensitivity-adjusted variable, and a residual (error term to account for unexplained variations). Consider [Table 3-3](#).

Table 3-3. Predicting y given x

y	x
100	49
200	99
300	149
400	199
?	249

The linear equation to predict y given x is as follows:

$$y_i = 2 + 2x_i$$

Therefore, the latest y given $x = 249$ should be 500:

$$y_i = 2 + 2x_i = 2 + (2 \times 249) = 500$$

Notice how linear regression perfectly captures the linear relationship between the two variables since there is no residual (unexplained variations). When a linear regression perfectly captures the relationship between two variables, it means that their coordinate points are perfectly aligned on a linear line across the x -axis.

A multiple linear regression can take the following form:

$$y_i = \alpha + \beta_1 x_1 + \dots + \beta_n x_n + \epsilon_i$$

This basically means that the dependent variable y may be impacted by more than one variable. For instance, if you want to estimate housing prices, you may want to take into account the number of rooms, the surface area, the neighborhood, and any other variable that is likely to impact the price. Similarly, if you want to predict commodity prices, you may want to take into account the different macroeconomic factors, exchange rates, and any other alternative data.

It is important to understand what every variable refers to. Linear regression makes a few assumptions:

Linear relationship

The relationship between the dependent variable and the independent variable(s) should be linear, meaning that a straight line across the plane can describe the relationship. This is rare in real life when dealing with complex variables.

Independence of variables

The observations should be independent of each other, meaning that the value of one observation does not influence the value of another observation.

Homoscedasticity

The variance of the residuals (the difference between the predicted and actual values of the dependent variable) should be constant across all levels of the independent variable(s).

Normality of the residuals

The residuals should be normally distributed, meaning that the majority of the residuals are close to zero and the distribution is symmetrical.

In the case of a multiple linear regression, you can add a new assumption: the absence of *multicollinearity*. The independent variables should not be highly correlated with each other; otherwise, it can make it difficult to determine the unique effect of each independent variable on the dependent variable. In other words, this prevents redundancy.



The key takeaways from this section are the following:

- Linear regression is part of the inferential statistics field, and it is a linear equation that describes the relationship between variables.
- Linear regression interprets and predicts data following an equation that you obtain when you train past data and expect the relationship to hold in the future.

Summary

Being able to perform data analysis is key to deploying the right algorithms so that you can predict the future values of a time series. Understanding data is done through a wide selection of tools coming from the statistics world. Make sure that you understand what stationarity and correlation are, as they offer extremely valuable insights in modeling.

CHAPTER 4

Linear Algebra and Calculus for Deep Learning

Algebra and calculus are integral parts of data science. Machine learning and deep learning algorithms are mostly based on algebra and calculus techniques. This chapter introduces some key topics in a way that everyone can understand.

Algebra is the study of operations and relational rules, as well as the constructions and ideas that result from them. Algebra covers topics such as linear equations and matrices. You can consider algebra as the first step toward calculus.

Calculus is the study of curve slopes and rates of change. Calculus covers topics such as derivatives and integrals. It is heavily used in many fields such as economics and engineering. Many learning algorithms rely on the concepts of calculus to perform their complex operations.

The distinction between the two is that while calculus works with ideas of change, motion, and accumulation, algebra deals with mathematical symbols and the rules for manipulating those symbols. Calculus focuses on the characteristics and behavior of changing functions, while algebra offers the foundation for solving equations and comprehending functions.

Linear Algebra

Algebra encompasses various mathematical structures, including numbers, variables, and operations like addition, subtraction, multiplication, and division. *Linear algebra* is a fundamental branch of algebra that deals with vector spaces and linear transformations. It is heavily used in machine learning and deep learning for tasks such as data preprocessing, dimensionality reduction, and solving systems of linear equations. Matrices and vectors are central data structures in linear algebra, and operations like matrix multiplication are common in various algorithms.

Vectors and Matrices

A *vector* is an object that has a magnitude (length) and a direction (arrowhead). The basic representation of a vector is an arrow with coordinates on the axis. But first, let's see what an axis is.

The *x*-axis and *y*-axis are perpendicular lines that specify a plane's boundaries and the locations of different points within them in a two-dimensional Cartesian coordinate system. The *x*-axis is horizontal and the *y*-axis is vertical.

These axes may represent vectors, with the *x*-axis representing the vector's horizontal component and the *y*-axis representing its vertical component.



In time series analysis, the *x*-axis is typically the time step (hours, days, etc.), and the *y*-axis is the value at the respective time step (price, return, etc.).

Figure 4-1 shows a simple two-dimensional Cartesian coordinate system with both axes.

The two-dimensional Cartesian coordinate system uses simple parentheses to show the location of different points following this order:

Point coordinates = (x, y)

The variable *x* represents the horizontal location

The variable *y* represents the vertical location

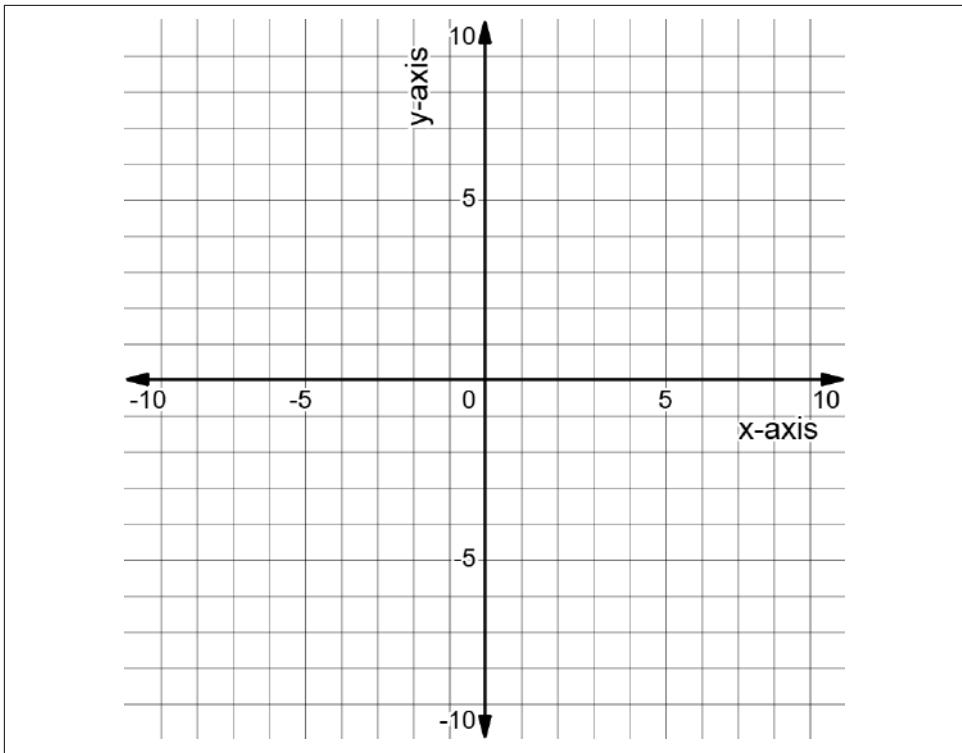


Figure 4-1. A two-dimensional Cartesian coordinate system

Therefore, if you want to draw point A, which has $(2, 3)$ as coordinates, you are likely to look at a graph from point zero, move two points to the right, and from there, move three points upward. The result of the point should look like [Figure 4-2](#).

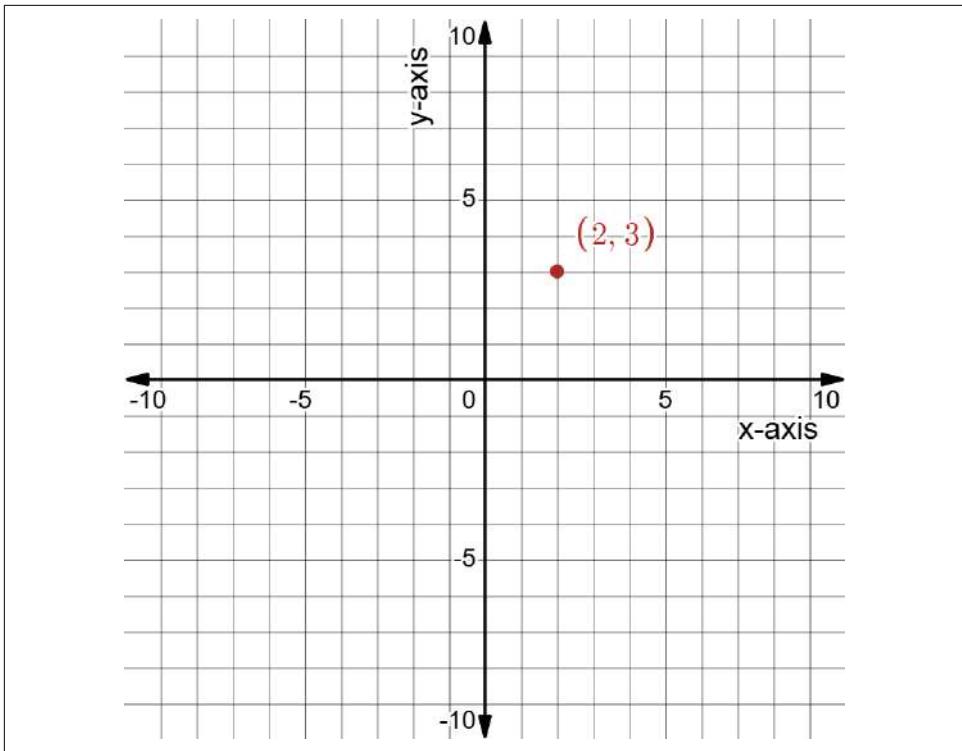


Figure 4-2. The location of A on the coordinate system

Let's now add another point and draw a vector between them. Suppose you have point B with (4, 5) as coordinates. Naturally, as the coordinates of B are higher than the coordinates of A, you would expect vector AB to be upward sloping. [Figure 4-3](#) shows the new point B and vector AB.

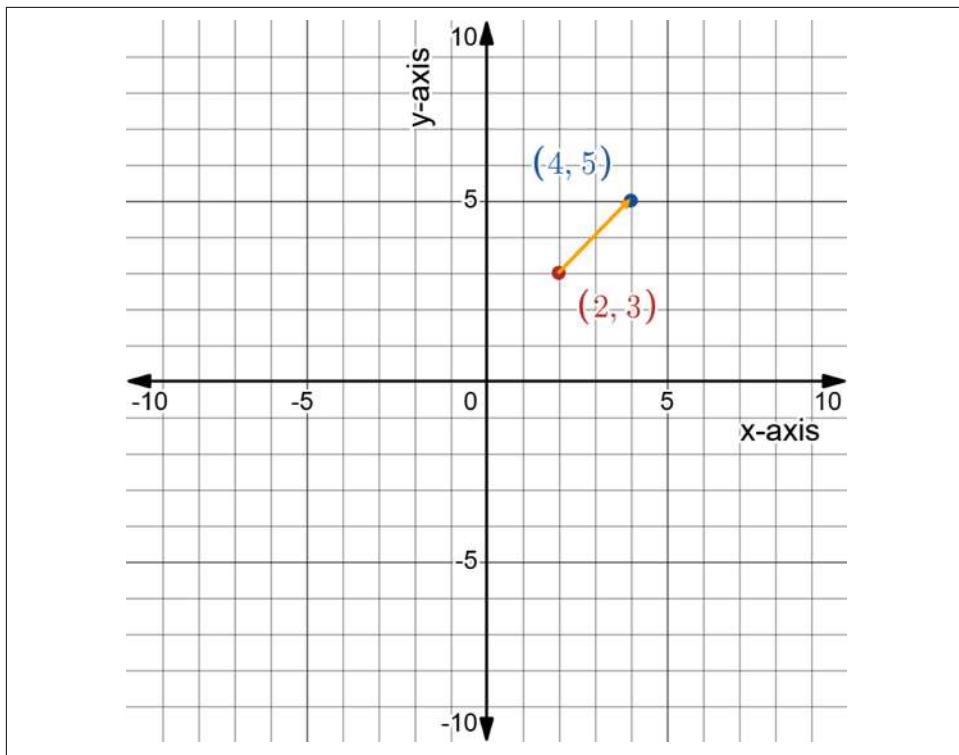


Figure 4-3. Vector AB joining points A and B together in magnitude and direction

However, having drawn the vector using the coordinates of both points, how would you refer to the vector? Simply put, vector AB has its own coordinates that represent it. Remember that the vector is a representation of the movement from point A to point B . This means the two-point movement along the x -axis and the y -axis is the vector. Mathematically, to find the vector, you should subtract the two coordinate points from each other while respecting the direction. Here's how to do that:

- *Vector AB* means that you are going from A to B ; therefore, you need to subtract the coordinates of point B from the coordinates of point A :

$$\overrightarrow{AB} = \langle 4 - 2, 5 - 3 \rangle$$

$$\overrightarrow{AB} = \langle 2, 2 \rangle$$

- *Vector BA* means that you are going from B to A; therefore, you need to subtract the coordinates of point A from the coordinates of point B:

$$\overrightarrow{BA} = \langle 2 - 4, 3 - 5 \rangle$$

$$\overrightarrow{BA} = \langle -2, -2 \rangle$$

To interpret the AB and BA vectors, you need to think in terms of movement. Vector AB represents going from point A to point B, two positive points horizontally and vertically (to the right and upward, respectively). Vector BA represents going from point B to point A, two negative points horizontally and vertically (to the left and downward, respectively).



Vectors AB and BA are not the same thing even though they share the same slope. But what is a slope anyway?

The *slope* is the ratio of the vertical change between two points on the line to the horizontal change between the same two points. You calculate the slope using this mathematical formula:

$$\text{Slope} = \frac{(\Delta Y)}{(\Delta X)}$$

$$\text{Slope of } \overrightarrow{AB} = \frac{2}{2} = 1$$

$$\text{Slope of } \overrightarrow{BA} = \frac{-2}{-2} = 1$$

If the two vectors were simply lines (with no direction), then they would be the same object. However, adding the directional component makes them two distinguishable mathematical objects.

Figure 4-4 sheds more light on the concept of the slope, as x has shifted two points to the right and y has shifted two points to the left.

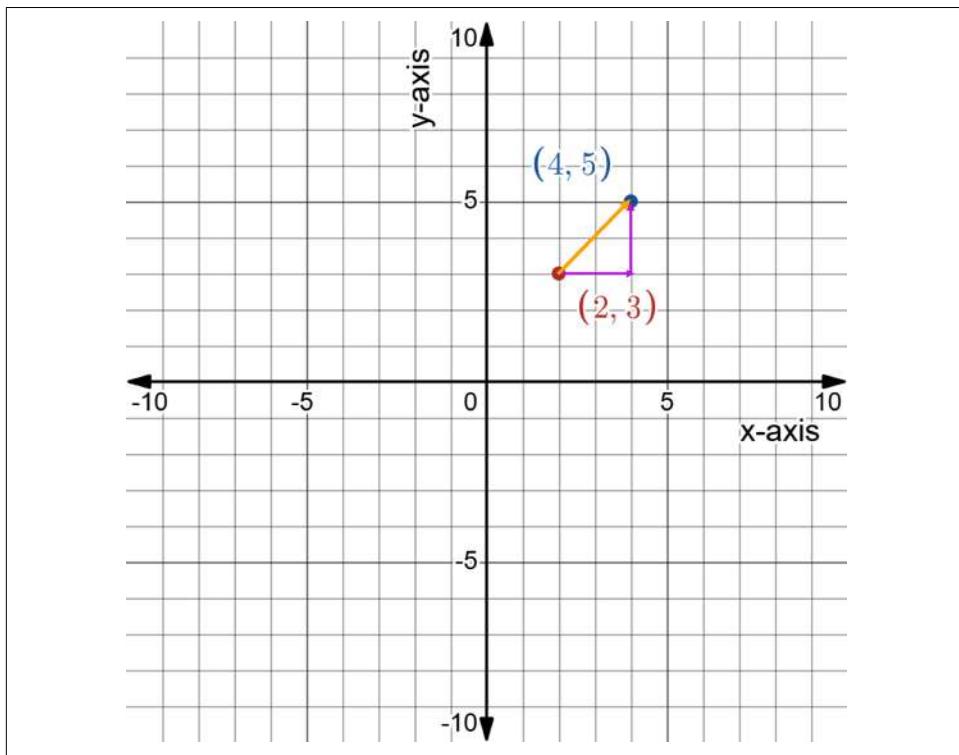


Figure 4-4. The change in x and the change in y for vector AB



A vector that has a magnitude of 1 is referred to as a *unit vector*.

Figure 4-5 shows the change in x and the change in y in the case of vector BA .

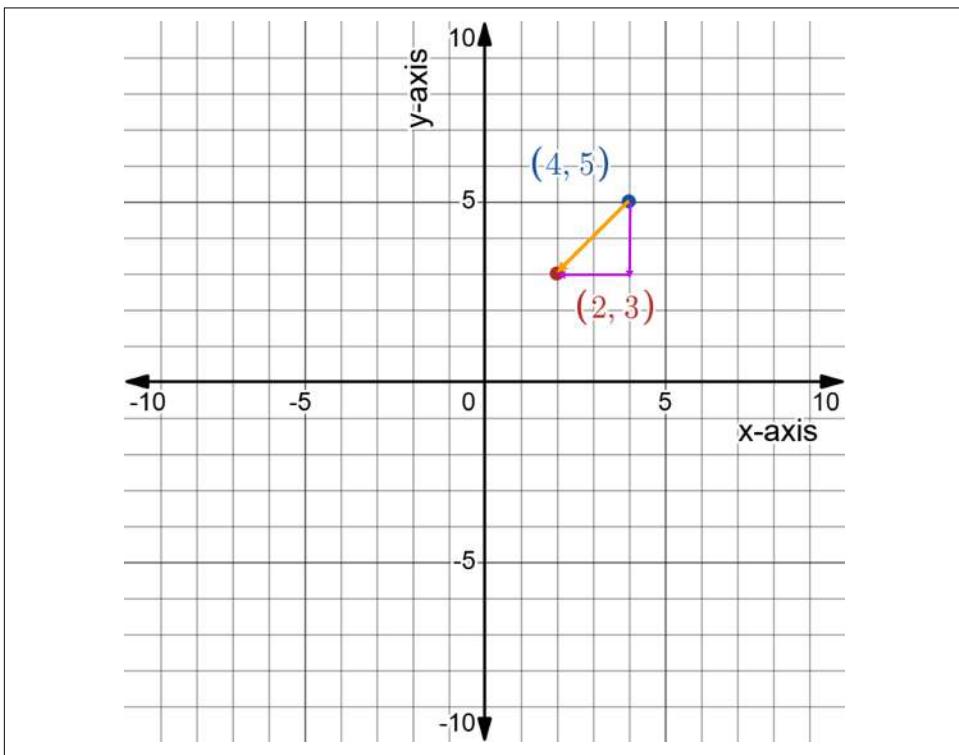


Figure 4-5. The change in x and the change in y for vector BA

Researchers typically use vectors as representations of speed, especially in engineering. Navigation is one field that heavily relies on vectors. It allows navigators to determine their positions and plan their destinations. Naturally, magnitude represents speed and the direction represents the destination.

You can add and subtract vectors from each other and from scalars. This allows for a shift in direction and magnitude. What you should retain from the previous discussion is that vectors indicate directions between different points on the axis.



A *scalar* is a value with magnitude but no direction. Scalars, as opposed to vectors, are used to represent elements, like temperature and prices. Basically, scalars are numbers.

A *matrix* is a rectangular array containing numbers and organized in rows and columns.¹ Matrices are useful in computer graphics and other domains as well as to define and manipulate linear systems of equations. What differentiates a matrix from a vector? The simplest answer is that a vector is a matrix with a single column or a single row. Here's a basic example of a 3×3 matrix:

$$\begin{bmatrix} 5 & 2 & 9 \\ -8 & 10 & 13 \\ 1 & 5 & 12 \end{bmatrix}$$

The size of a matrix is the number of rows and columns it contains. A row is a horizontal line, and a column is a vertical line. The following representation is a 2×4 matrix (i.e., two rows by four columns):

$$\begin{bmatrix} 5 & 2 & 1 & 3 \\ -8 & 10 & 9 & 4 \end{bmatrix}$$

The following representation is a 4×2 matrix (i.e., four rows by two columns):

$$\begin{bmatrix} 5 & 2 \\ -8 & 10 \\ 8 & 22 \\ 7 & 3 \end{bmatrix}$$



Matrices are heavily used in machine learning. Rows generally represent time and columns represent features.

The summation of different matrices is straightforward but must be used only when the matrices match in size (which means they have the same number of columns and rows). For instance, let's add the following two matrices:

$$\begin{bmatrix} 1 & 2 \\ 5 & 8 \end{bmatrix} + \begin{bmatrix} 3 & 9 \\ 1 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 11 \\ 6 & 13 \end{bmatrix}$$

¹ Matrices can also contain symbols and expressions, but for the sake of simplicity, let's stick to numbers.

You can see that to add two matrices, you simply have to add the numbers in the same positions. Now, if you try to add the next pair of matrices, you won't be able to do it as there is a mismatch in what to add:

$$\begin{bmatrix} 8 & 3 \\ 3 & 2 \end{bmatrix} + \begin{bmatrix} 3 & 9 \\ 1 & 5 \\ 5 & 4 \end{bmatrix}$$

The subtraction of matrices is also straightforward and follows the same rules as the summation of matrices. Let's take the following example:

$$\begin{bmatrix} 5 & 2 \\ -8 & 10 \end{bmatrix} - \begin{bmatrix} 3 & 9 \\ -1 & -5 \end{bmatrix} = \begin{bmatrix} 2 & -7 \\ -9 & 15 \end{bmatrix}$$

Evidently, subtraction of matrices is also a summation of matrices with a change of signs in one of them.

Matrix multiplication by a scalar is quite simple. Let's take the following example:

$$3 \times \begin{bmatrix} 5 & 2 \\ 8 & 22 \end{bmatrix} = \begin{bmatrix} 15 & 6 \\ 24 & 66 \end{bmatrix}$$

So basically, you are multiplying every cell in the matrix by the scalar. Multiplying one matrix by another matrix is a bit more complicated as it uses the *dot product* method. First of all, to multiply two matrices together, they must satisfy this condition:

$$\text{Matrix}_{xy} \times \text{Matrix}_{yz} = \text{Matrix}_{xz}$$

This means that the first matrix must have a number of columns equal to the number of rows in the second matrix, and the resulting matrix from the dot product is a matrix that has the number of rows of the first matrix and the number of columns of the second matrix. The dot product is explained in the following example representation of a 1×3 and 3×1 matrix multiplication (notice the equal number of columns and rows):

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = [(1 \times 3) + (2 \times 2) + (3 \times 1)] = [10]$$

Now let's take an example of a 2×2 matrix multiplication:

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ 2 & 1 \end{bmatrix}$$

There is a special type of matrix called the *identity matrix*, which is basically the number 1 for matrices. It is defined as follows for a 2×2 dimension:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and as follows for a 3×3 dimension:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying any matrix by the identity matrix yields the same original matrix. This is why it can be referred to as the 1 of matrices (multiplying any number by 1 yields the same number). It is worth noting that matrix multiplication is not commutative, which means that the order of multiplication changes the result:

$$AB \neq BA$$

Matrix transposing is a process that involves changing the rows into columns and vice versa. The transpose of a matrix is obtained by reflecting the matrix along its main diagonal:

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

Transposing is used in some machine learning algorithms and is not an uncommon operation when dealing with such models. If you are wondering about the role of matrices in data science and machine learning, you can refer to this nonexhaustive list:

Representation of data

Matrices often represent data with rows representing samples and columns representing features. For example, a row in a matrix can present OHLC data in one time step.

Linear algebra

Matrices and linear algebra are intertwined, and many learning algorithms use the concepts of matrices in their operations. Having a basic understanding of these mathematical concepts helps smooth the learning curve when dealing with machine learning algorithms.

Data relationship matrices

Covariance and correlation measures are often represented as matrices. These relationship calculations are important concepts in time series analysis.



The key takeaways from this section are as follows:

- A vector is an object that has a magnitude (length) and a direction (arrowhead). Multiple vectors grouped together form a matrix.
- A matrix can be used to store data. It has its special ways of performing operations.
- Matrix multiplication uses the dot product method.
- Transposing a matrix means to swap its rows and its columns.

Introduction to Linear Equations

You saw an example of a linear equation in “[Regression Analysis and Statistical Inference](#)” on page 77. *Linear equations* are basically formulas that present an equality relationship between different variables and constants. In the case of machine learning, it is often a relationship between a dependent variable (the output) and an independent variable (the input). The best way to understand linear equations is through examples.



The aim of linear equations is to find an unknown variable, usually denoted by the letter x .

We'll start with a very basic example that you can consider as a first building block toward the more advanced concepts you will see later on. The following example requires finding the value of x that satisfies the equation:

$$10x = 20$$

You should understand the equation as "*10 times which number equals 20?*" When a constant is directly attached to a variable such as x , it refers to a multiplication operation. Now, to solve for x (i.e., finding the value of x that equalizes the equation), you have an obvious solution, which is to get rid of 10 so that you have x on one side of the equation and the rest on the other side.

Naturally, to get rid of 10, you divide by 10 so that what remains is 1, which if multiplied by the variable x does nothing. However, keep in mind two important things:

- If you do a mathematical operation on one side of an equation, you must do it on the other side as well. This is why they are called equations.
- For simplicity, instead of dividing by the constant to get rid of it, you should multiply it by its reciprocal.

The *reciprocal* of a number is 1 divided by that number. Here's the mathematical representation of it:

$$\text{Reciprocal}(x) = \frac{1}{x}$$

Now, back to the example, to find x you can do the following:

$$\left(\frac{1}{10}\right)10x = 20\left(\frac{1}{10}\right)$$

Performing the multiplication and simplifying gives the following result:

$$x = 2$$

This means that the solution of the equation is 2. To verify this, you just need to plug 2 into the original equation as follows:

$$10 \times 2 = 20$$

Therefore, it takes two 10s to get 20.



Dividing the number by itself is the same thing as multiplying it by its reciprocal.

Let's take another example of how to solve x through linear techniques. Consider the following problem:

$$\frac{8}{6}x = 24$$

Performing the multiplication and simplifying gives the following result:

$$\left(\frac{6}{8}\right)\frac{8}{6}x = 24\left(\frac{6}{8}\right)$$

$$x = 18$$

This means that the solution of the equation is 18. To verify this, you just need to plug 18 into the original equation as follows:

$$\frac{8}{6} \times 18 = 24$$

Typically, linear equations are not this simple. Sometimes they contain more variables and more constants, which need more detailed solutions, but let's keep taking it step by step. Consider the following example:

$$3x - 6 = 12$$

Solving for x requires rearranging the equation a little bit. Remember, the aim is to leave x on one side and the rest on the other. Here, you have to get rid of the constant 6 before taking care of 3. The first part of the solution is as follows:

$$3x - 6(+6) = 12(+6)$$

Notice how you have to add 6 to both parts of the equation. The part on the left will cancel itself out, while the part on the right will add up to 18:

$$3x = 18$$

Finally, you're all set to multiply by the reciprocal of the constant attached to the variable x :

$$\left(\frac{1}{3}\right)3x = 18\left(\frac{1}{3}\right)$$

Simplifying and solving for x leaves the following solution:

$$x = 6$$

This means that the solution of the equation is 6. To verify this, just plug 6 into the original equation as follows:

$$(3 \times 6) - 6 = 12$$

By now, you should have noticed that linear algebra is all about using shortcuts and quick techniques to simplify equations and find unknown variables. The next example shows how sometimes the variable x can occur in multiple places:

$$6x + x = 27 - 2x$$

Remember, the main focus is to have x on one side of the equation and the rest on the other side:

$$6x + x + 2x = 27$$

Adding the constants of x gives you the following:

$$9x = 27$$

The final step is dividing by 9 so that you only have x remaining:

$$x = 3$$

You may now verify this by plugging 3 in the place of x in the original equation. You will notice that both sides of the equation will be equal.



Even though this section is quite simple, it contains the basic foundations you need to start advancing in algebra and calculus. The key takeaways from this section are as follows:

- A linear equation is a representation in which the highest exponent on any variable is one. This means that there are no variables that are raised to the power of two and above.
- A linear equation line is straight when plotted on a chart.
- The application of linear equations in modeling a wide range of real-world occurrences makes them crucial in many branches of mathematics and research. They are also widely utilized in machine learning.
- Solving for x is the process of finding for it a value that equalizes both sides of the equation.
- When performing an operation (such as adding a constant or multiplying by a constant) on one side of the equation, you have to do it on the other side as well.

Systems of Equations

A *system of equations* is when there are two or more equations working together to solve one or more variables. Therefore, instead of the usual single equation:

$$x + 10 = 20$$

systems of equations resemble the following:

$$x + 10 = 20$$

$$y + 2x = 10$$

Systems of equations are useful in machine learning and are used in many of its aspects.

Let's look at the previous system of equations from the beginning of this section and solve it graphically. Plotting the two functions can actually give the solution directly. The point of intersection is the solution. Therefore, the coordinates of the intersection (x, y) refer to the solutions of x and y , respectively.

From [Figure 4-6](#), it seems that $x = 10$ and $y = -10$. Plugging these values into their respective variables gives the correct answer:

$$10 + 10 = 20$$

$$(-10) + (2 \times 10) = 10$$

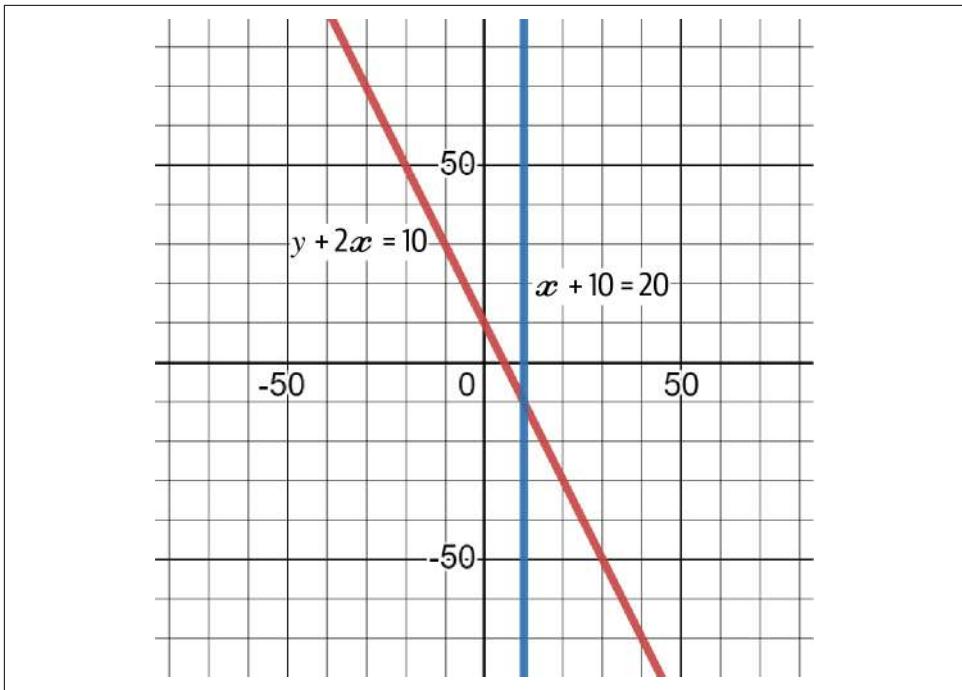


Figure 4-6. A graph showing the two functions and their intersection (solution)

As the functions are linear, solving them can result in one of three outcomes:

1. There is only one solution for each variable.
2. There is no solution. This occurs when the functions are *parallel* (this means that they never intersect).
3. There are an infinite number of solutions. This occurs when, through simplification, both functions are the same (since all points fall on the straight line).

Before moving on to solving systems of equations using algebra, let's visually see how there can be no solution and how there can be an infinite number of solutions. Consider the following system:

$$2x = 10$$

$$4x = 20$$

[Figure 4-7](#) charts the two together. Since they are exactly the same equation, they fall on the same line. In reality, there are two lines in [Figure 4-7](#), but since they are the same, they are indistinguishable. For every x on the line, there is a corresponding y .

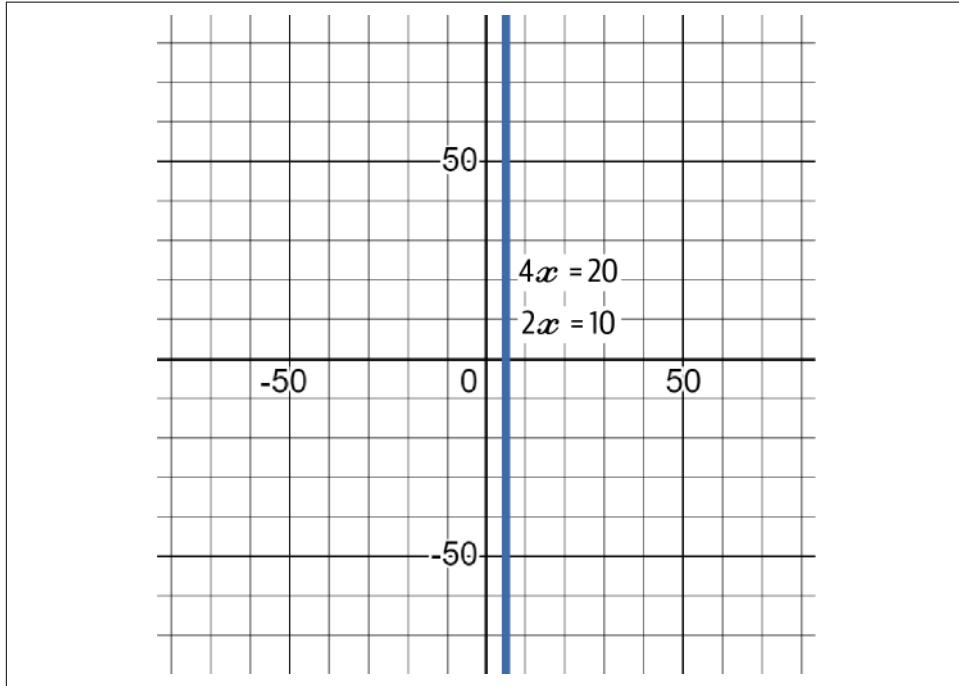


Figure 4-7. A graph showing the two functions and their infinite intersections

Now consider the following system:

$$3x = 10$$

$$6x = 10$$

[Figure 4-8](#) shows how they never intersect, which is intuitive as you cannot multiply the same number (represented by the variable x) with different numbers and expect to get the same result.

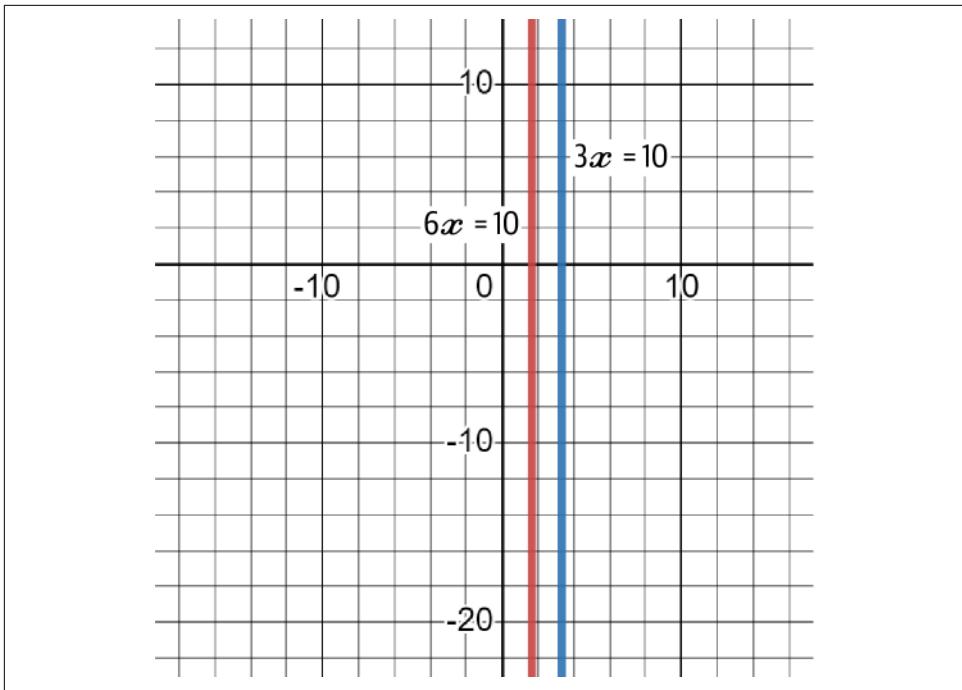


Figure 4-8. A graph showing the two functions and their impossible intersection

Algebraic methods are used when there are more than two variables since they cannot be solved through graphs. This mainly entails two methods: substitution and elimination.

Substitution is used when you can replace the value of a variable in one equation and plug it into the second equation. Consider the following example:

$$x + y = 2$$

$$10x + y = 10$$

The easiest method is to rearrange the first equation so that you have y in terms of x :

$$y = 2 - x$$

$$10x + (2 - x) = 10$$

Solving for x in the second equation becomes simple:

$$10x + (2 - x) = 10$$

$$10x + 2 - x = 10$$

$$10x - x = 10 - 2$$

$$9x = 8$$

$$x = \frac{8}{9}$$

$$x = 0.8889$$

Now that you have found the value of x , you can easily find y by plugging the value of x into the first equation:

$$0.8889 + y = 2$$

$$y = 2 - 0.8889$$

$$y = 1.111$$

To check if your solution is correct, you can plug in the values of x and y in both formulas:

$$0.8889 + 1.111 = 2$$

$$(10 \times 0.8889) + 1.111 = 10$$

Graphically, this means that the two equations intersect at $(0.8889, 1.111)$. This technique can be used with more than two variables. Follow the same process until the equations are simplified enough to give you the answers. The issue with substitution is that it may take some time when you're dealing with more than two variables.

Elimination is a faster alternative. It is about eliminating variables until there is only one left. Consider the following example:

$$2x + 4y = 20$$

$$3x + 2y = 10$$

Noticing that there is $4y$ and $2y$, it is possible to multiply the second equation by 2 so that you can subtract the equations from each other (which will remove the y variable):

$$2x + 4y = 20$$

$$6x + 4y = 20$$

Subtracting the two equations from each other gives the following result:

$$-4x = 0$$

$$x = 0$$

Therefore, $x = 0$. Graphically, this means that they intersect whenever $x = 0$ (exactly at the vertical y line). Plugging the value of x into the first formula gives $y = 5$:

$$(2 \times 0) + 4y = 20$$

$$4y = 20$$

$$y = 5$$

Similarly, elimination can also solve equations with three variables. The choice between substitution and elimination depends on the type of equation being solved.



The key takeaways from this section are as follows:

- Systems of equations solve variables together. They are very useful in machine learning and are used in some algorithms.
- Graphical solutions are preferred for simple systems of equations.
- Solving systems of equations through algebra entails the use of substitution and elimination methods.
- Substitution is preferred when the system is simple, but elimination is the way to go when the system is a bit more complex.

Trigonometry

Trigonometry explores the behavior of what is known as *trigonometric functions* that relate the angles of a triangle to the lengths of its sides. The most-used triangle is the right-angled triangle, which has one angle at 90° . [Figure 4-9](#) shows an example of a right-angled triangle.

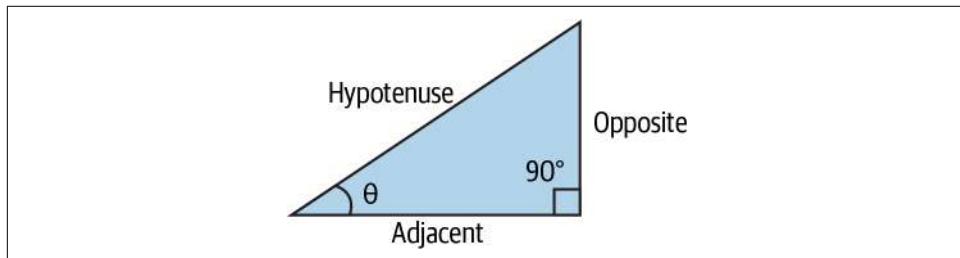


Figure 4-9. A right-angled triangle

Let's define the main characteristics of a right-angled triangle:

- The longest side of the triangle is called the *hypotenuse*.
- The angle in front of the hypotenuse is the right angle (the one at 90°).
- Depending on the other angle (θ) you choose (from the two that remain), the line between this angle and the hypotenuse is called the *adjacent* and the other line is called the *opposite*.



Trigonometric functions are mathematical functions used to relate the angles of a right triangle to the ratios of its sides. They have various applications in fields like geometry, physics, engineering, and more. They help analyze and solve problems related to angles, distances, oscillations, and waveforms, among other things.

Trigonometric functions are simply the division of one line by another line. Remember that you have three lines in a triangle (hypotenuse, opposite, and adjacent). The trigonometric functions are found as follow:

$$\sin(\theta) = \frac{\text{Opposite}}{\text{Hypotenuse}}$$

$$\cos(\theta) = \frac{\text{Adjacent}}{\text{Hypotenuse}}$$

$$\tan(\theta) = \frac{\text{Opposite}}{\text{Adjacent}}$$

From the previous three trigonometric functions, it is possible to extract a trigonometric identity that reaches *tan* from *sin* and *cos* using basic linear algebra:

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}$$

Hyperbolic functions are similar to trigonometric functions but are defined using exponential functions. Before understanding hyperbolic functions, one must understand Euler's number.



This part on hyperbolic functions is interesting as it forms the basis of what is known as *activation functions*, a key concept in neural networks, the protagonists of deep learning models. You will see them in detail in [Chapter 8](#).

Euler's number (denoted as e) is one of the most important numbers in mathematics. It is an *irrational number*, which is a real number that cannot be expressed as a fraction. The word *irrational* comes from the fact that there is no *ratio* to express it; it has nothing to do with its personality. Euler's number is also the base of the natural logarithm \ln , and the first digits of it are 2.71828. One of the best approximations to get e is the following formula:

$$e = \left(1 + \frac{1}{n}\right)^n$$

By increasing n in the previous formula, you will approach the value of e . Euler's number has many interesting properties, most notably the fact that its slope is its own value. Consider the following function (also called the *natural exponent function*):

$$f(x) = e^x$$

At any point, the slope of the function is the same value. Take a look at [Figure 4-10](#).

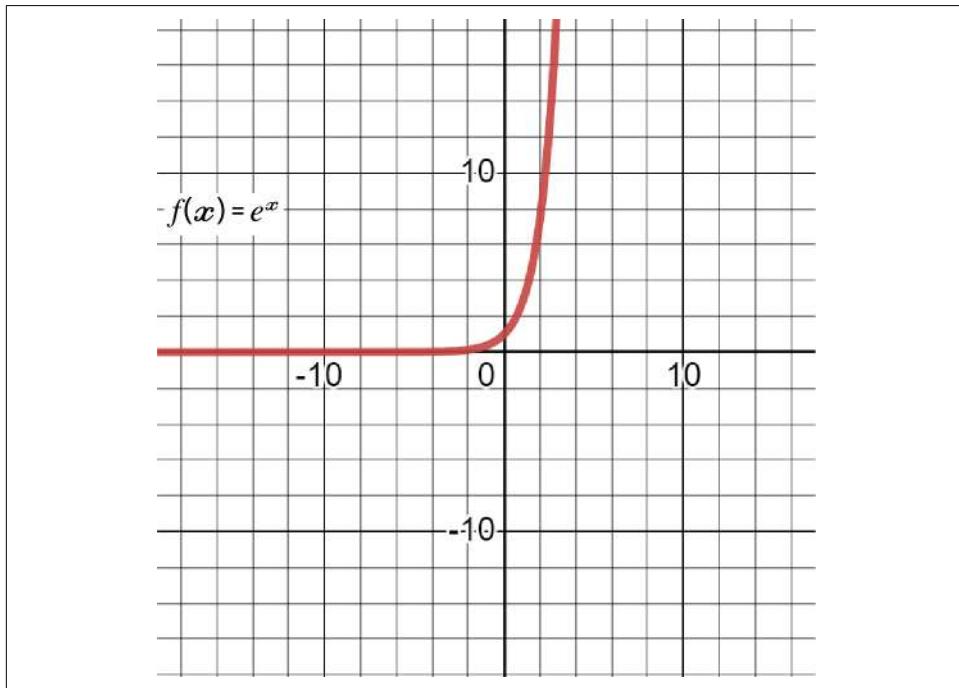


Figure 4-10. A graph of the natural exponent function



You may be wondering why I am explaining exponents and logarithms in this book. There are mainly two reasons for this:

- Exponents and, more importantly, Euler's number are used in hyperbolic functions where $\tanh(x)$ is one of the main activation functions for neural networks, a type of machine and deep learning model.
- Logarithms are useful in *loss functions*, a concept that you will see in later chapters.

Hyperbolic functions use the natural exponent function and are defined as follows:

$$\sinh(x) = \frac{e^x - e^{-x}}{2}$$

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Among the key characteristics of $\tanh(x)$ are nonlinearity, the limitation between $[-1, 1]$, and the fact that it is centered at zero. [Figure 4-11](#) shows the graph of $\tanh(x)$.

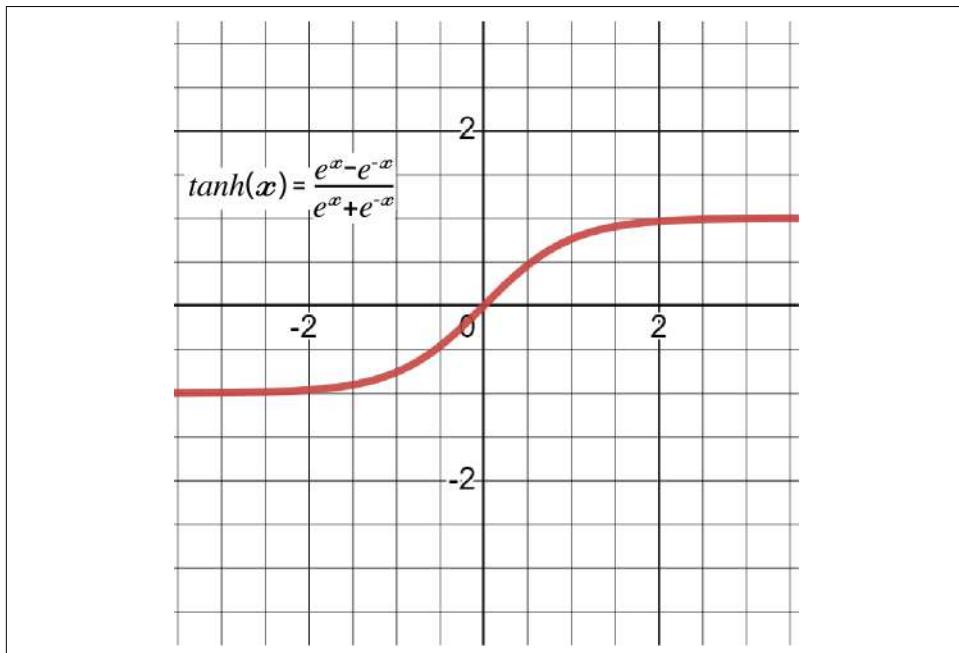


Figure 4-11. A graph of $\tanh(x)$ showing how it's limited between -1 and 1



The key takeaways from this section are as follows:

- Trigonometry is a field that explores the behavior of trigonometric functions that relate the angles of a triangle to the lengths of its sides.
- A trigonometric identity is a shortcut that relates the trigonometric functions with each other.
- Euler's number e is irrational and is the base of the natural logarithm. It has many applications in exponential growth and in hyperbolic functions.
- The hyperbolic tangent function is used in neural networks, a deep learning algorithm.

Calculus

As previously mentioned, calculus is a branch of mathematics that focuses on the study of rates of change and accumulation of quantities. It consists of two primary branches: *differential calculus* (which deals with derivatives) and *integral calculus* (which deals with integration). This section briefly introduces both types of calculus while also discussing topics such as limits and optimization.

Limits and Continuity

Calculus works by making visible the infinitesimally small.

—Keith Devlin

Limits don't have to be nightmarish. I have always found them to be misunderstood. They are actually quite easy to get. But first, you need motivation, and this comes from knowing the added value of learning limits.

Understanding limits is important in machine learning models for many reasons:

Optimization

In optimization methods like gradient descent, limits can be used to regulate the step size and guarantee convergence to a local minimum.

Feature selection

Limits can be used to rank the significance of various model features and perform feature selection, which can make the model simpler and perform better.

Sensitivity analysis

A machine learning model's sensitivity to changes in input data and its capacity to generalize to new data can be used to examine a model's behavior.

Also, limits are used in more advanced calculus concepts that you will learn about shortly.

The main aim of limits is to know the value of a function when it's undefined. But what is an *undefined function*? When you have a function that gives a solution that is not possible (such as dividing by zero), limits help you bypass this issue in order to know the value of the function at that point. So the aim of limits is to solve functions even when they are undefined.

Remember that the solution to a function that takes x as an input is a value in the y -axis. **Figure 4-12** shows a linear graph of the following function:

$$f(x) = x + 2$$

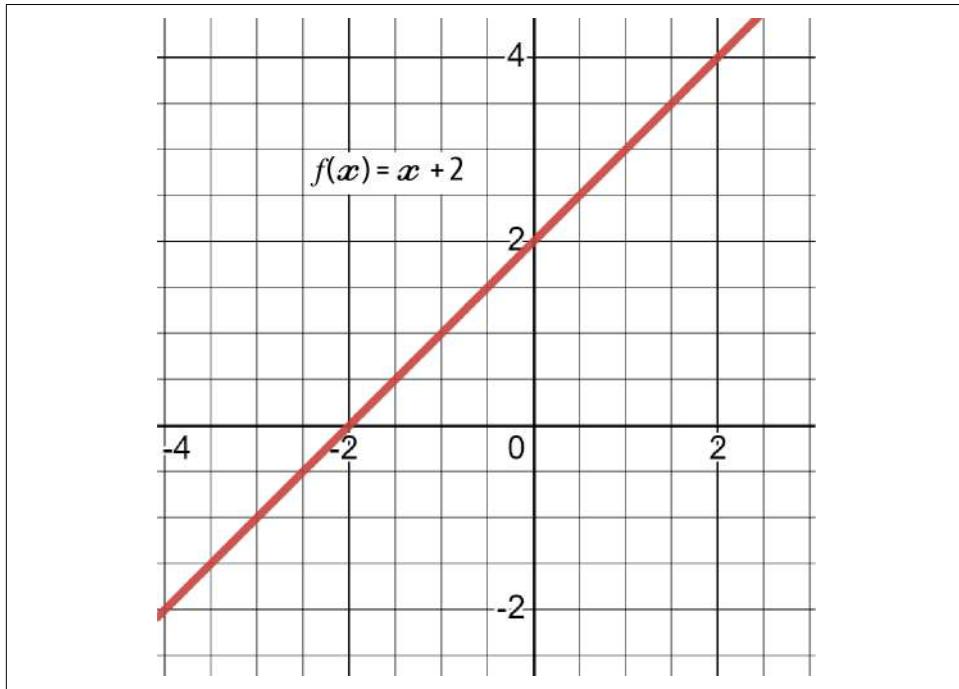


Figure 4-12. A graph of the function $f(x) = x + 2$

The solution of the function in the graph is the one that lies on the linear line taking into account the value of x every time.

What would be the solution of the function (the value of y) when $x = 4$? Clearly, the answer is 6, as substituting the value of x with 4 gives 6:

$$f(4) = 4 + 2 = 6$$

Thinking of this solution in terms of limits would be like asking for the solution of the function as x approaches 4 from both sides (the negative/decreasing side and the positive/increasing side). **Table 4-1** simplifies this dilemma.

Table 4-1. Finding x as it approaches 4

$f(x)$	x
5.998	3.998
5.999	3.999
6.000	4.000
6.001	4.001
6.002	4.002

Approaching from the negative side is the equivalent of adding a fraction of a number while below 4 and analyzing the result every time. Similarly, approaching from the positive side is the equivalent of removing a fraction of a number while above 4 and analyzing the result every time. The solution seems to converge to 6 as x approaches 4. This is the solution to the limit.

Limits in the general form are written following this convention:

$$\lim_{x \rightarrow a} f(x) = L$$

The general form of the limit is read as follows: as you approach a along the x -axis (whether from the positive or the negative side), the function $f(x)$ gets closer to the value of L .



The idea of the limit states that as you lock in and approach a number from either side (negative or positive), the solution of the equation approaches a certain number, and the solution to the limit is that number.

As mentioned previously, limits are useful when the exact point of the solution is undefined using the conventional way of substitution.

A one-sided limit is different from the general limit. With a lefthand limit, you search for the limit going from the negative side to the positive side, and with a righthand limit, you search for the limit going from the positive side to the negative side. The general limit exists when the two one-sided limits exist and are equal. Therefore, the previous statements are summarized as follows:

- The lefthand limit exists.
- The righthand limit exists.
- The lefthand limit is equal to the righthand limit.

The lefthand limit is defined as follows:

$$\lim_{x \rightarrow a^-} f(x) = L$$

The righthand limit is defined as follows:

$$\lim_{x \rightarrow a^+} f(x) = L$$

Consider the following equation:

$$f(x) = \frac{x^3 - 27}{x - 3}$$

What is the solution of the function when $x = 3$? Substitution leads to the following issue:

$$f(3) = \frac{3^3 - 27}{3 - 3} = \frac{27 - 27}{3 - 3} = \frac{0}{0} = \text{Undefined}$$

However, thinking about this in terms of limits as shown in [Table 4-2](#), it seems that as you approach $x = 3$, either from the left or right side, the solution tends to approach 27.

Table 4-2. Finding x as it approaches 3

f(x)	x
2.9998	26.9982
2.9999	26.9991
3.0000	Undefined
3.0001	27.0009
3.0002	27.0018

Graphically, this can be seen as a discontinuity in the chart along both axes. The discontinuity exists on the line around the coordinate (3, 27). Some functions do not have limits. For example, what is the limit of the following function as x approaches 5?

$$\lim_{x \rightarrow 5} \frac{1}{x - 5}$$

Looking at **Table 4-3**, it seems that as x approaches 5, the results highly diverge when approaching from both sides. For instance, approaching from the negative side, the limit of 4.9999 is -10,000, and from the positive side, the limit of 5.0001 is 10,000.

Table 4-3. Finding x as it approaches 5

f(x)	x
4.9998	-5000
4.9999	-10000
5.0000	Undefined
5.0001	10000
5.0002	5000

Remember that for the general limit to exist, both one-sided limits must exist and must be equal, which is not the case here. Graphing this gives **Figure 4-13**, which may help you understand why the limit does not exist.

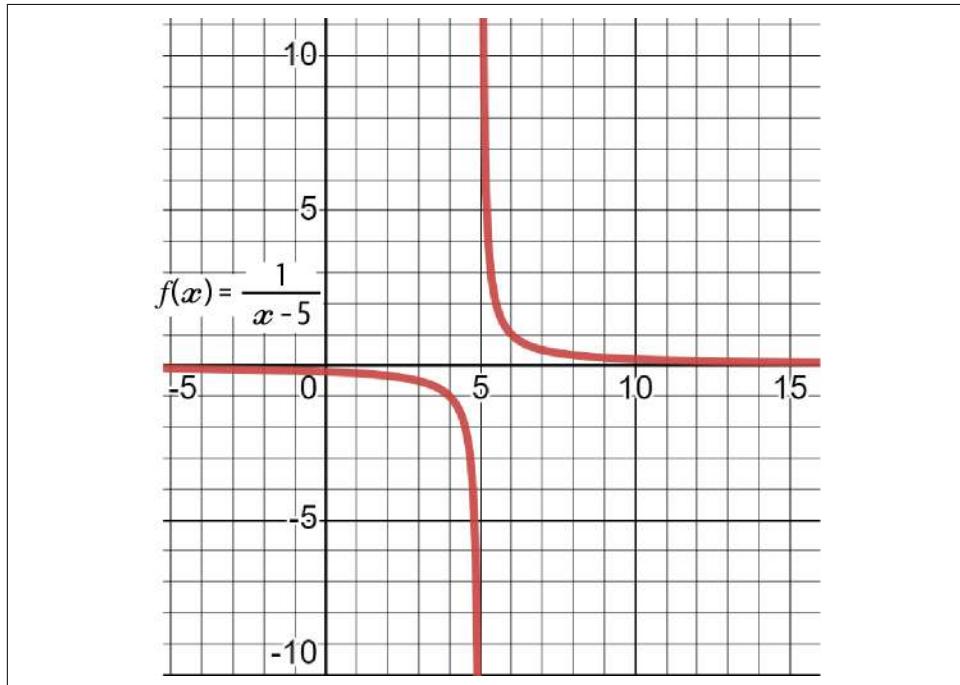


Figure 4-13. A graph of the function proving that the limit does not exist

But what if the function that you want to analyze looks like this:

$$\lim_{x \rightarrow 5} \frac{1}{|x - 5|}$$

Looking at [Table 4-3](#), it seems that as x approaches 5, the results rapidly accelerate as they diverge to a very big number referred to as infinity (∞):

$$f(x) = \frac{1}{|x - 5|}$$

Take a look at [Table 4-4](#):

Table 4-4. Another attempt at finding x as it approaches 5

f(x)	x
4.99997	334333.33
4.99998	50000
4.99999	100000
4.999999	10000000
5.00000	Undefined
5.0000001	10000000
5.00001	100000
5.00002	50000
5.00003	334333.33

At every tiny step, x approaches 5, and y approaches positive infinity. The answer to the limit question is therefore positive infinity ($+\infty$). [Figure 4-14](#) shows the graph of the function. Notice how both sides rise in value as x approaches 5.

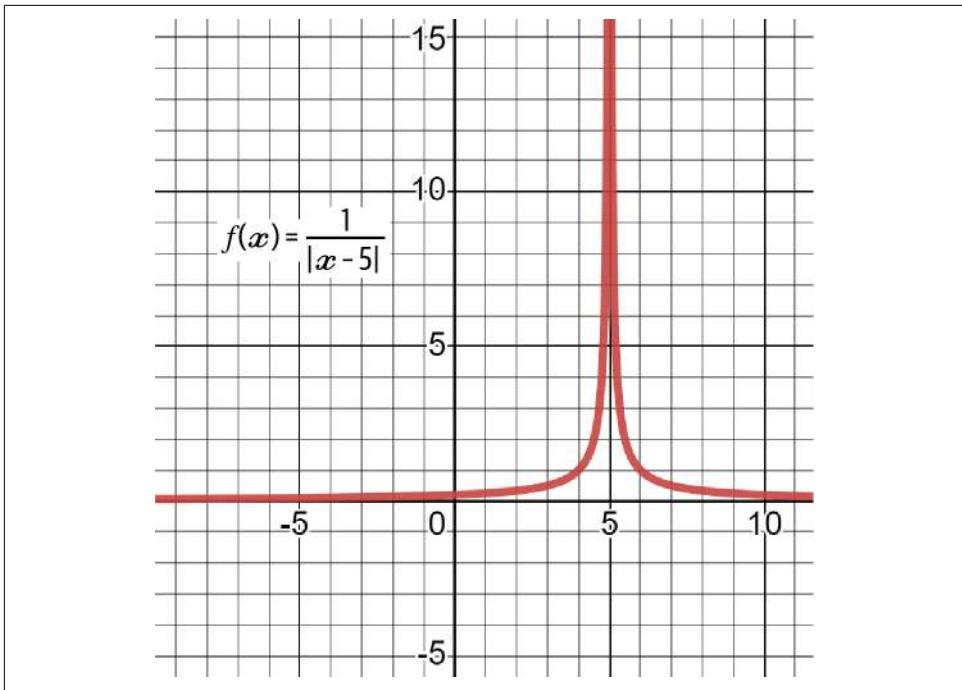


Figure 4-14. A graph of the function proving that the limit exists as x approaches 5

To Infinity and Beyond

You can understand what infinity represents in terms of mathematics. *Infinity* is an idea or a concept rather than a number. The symbol (∞) is often referred to as a *lemniscate*.

Positive infinity (∞) and negative infinity ($-\infty$) are both concepts that exist across the axis where the former tends toward the right and the latter tends toward the left.

Interestingly, infinity is often thought of as an ever-growing measure, but it is not expanding or getting bigger. It is already what it is.

Mathematical operations, including the concept of infinity, may be hard to grasp and have many considerations. One of the most interesting examples is the answer to why the result of 1 divided by zero is undefined.

Imagine dividing an apple among 10 people; normally every person will get an equal fraction ($1/10$) of that apple until the apple is consumed. But what if you want to divide the apple by zero people? The number of people required to consume the apple will tend toward infinity (the logic is hard to grasp, but mathematically you can think of infinitesimally small parts of the apple requiring a huge number of people to consume them).

Therefore, following this logic, you can say that 1 divided by 0 is infinity and is actually defined. So, why is it generally considered undefined? The issue is that the apple example describes positive infinity, and if you want to follow the example of dividing 1 by infinitesimally small negative numbers that tend toward zero, then you will also say that 1 divided by 0 is negative infinity.

So which is it? Positive or negative infinity? Because of this conflict, the result is undefined.

Continuous functions are ones that are drawn without gaps or holes in the graph, while *discontinuous* functions contain such gaps and holes. This usually means that the latter contain points where the solution of the functions is undefined and may need to be approximated by limits. Therefore, continuity and limits are two related concepts.

Let's proceed to solving limits; after all, you are not going to create a table every time and analyze the results subjectively to find the limits. There are three ways to solve limits:

- *Substitution*: This is the simplest rule and is generally used first.
- *Factoring*: This comes after substitution does not work.
- *Conjugate methods*: This solution comes after the first two do not work.

Substitution involves simply plugging in the value that x approaches. Basically, these are functions that have solutions where the limits are used. Take the following example:

$$\lim_{x \rightarrow 5} x + 10 - 2x$$

Using substitution, the limit of the function is found as follows:

$$\lim_{x \rightarrow 5} x + 10 - 2x = 5 + 10 - (2 \times 5) = 5$$

Therefore, the answer to the limit is 5.

Factoring is the next option when substitution does not work (e.g., the limit is undefined after plugging the value of x into the function). *Factoring* is all about changing the form of the equation using factors in such a way that the equation is not undefined anymore when using substitution. Take the following example:

$$\lim_{x \rightarrow -6} \frac{(x+6)(x^2-x+1)}{x+6}$$

If you try substitution, you will get an undefined value as follows:

$$\lim_{x \rightarrow -6} \frac{(x+6)(x^2-x+1)}{x+6} = \frac{(-6+6)((-6)^2 - (-6) + 1)}{-6+6} = \frac{0}{0} = \text{Undefined}$$

Factoring may help in this case. For example, the numerator is multiplied by $(x+6)$ and then divided by $(x+6)$. Simplifying this by canceling the two terms could give a solution:

$$\lim_{x \rightarrow -6} \frac{(x+6)(x^2-x+1)}{x+6} = \lim_{x \rightarrow -6} x^2 - x + 1$$

Now that factoring is done, you can try substitution once again:

$$\lim_{x \rightarrow -6} x^2 - x + 1 = (-6)^2 - (-6) + 1 = 43$$

The limit of the function as x tends toward -6 is therefore 43.

Forming a conjugate is the next option when substitution and factoring do not work. A *conjugate* is formed by simply changing signs between two variables. For example, the conjugate of $x+y$ is $x-y$. The way to do this in the case of a fraction is to multiply the numerator and the denominator by the conjugate of one of them (with a preference to use the conjugate of the term that has a square root since it will get canceled out). Consider the following example:

$$\lim_{x \rightarrow 9} \frac{x-9}{\sqrt{x}-3}$$

By multiplying both terms by the conjugate of the denominator, you will have started to use the conjugate method to solve the problem:

$$\lim_{x \rightarrow 9} \frac{x-9}{\sqrt{x}-3} \left(\frac{\sqrt{x}+3}{\sqrt{x}+3} \right)$$

Taking into account the multiplication and then simplifying gives the following:

$$\lim_{x \rightarrow 9} \frac{(x-9)(\sqrt{x}+3)}{(\sqrt{x}-3)(\sqrt{x}+3)}$$

You will be left with the following familiar situation:

$$\lim_{x \rightarrow 9} \frac{(x - 9)(\sqrt{x} + 3)}{x - 9}$$

$$\lim_{x \rightarrow 9} \sqrt{x} + 3$$

Now the function is ready for substitution:

$$\lim_{x \rightarrow 9} \sqrt{9} + 3 = 3 + 3 = 6$$

The solution to the function is therefore 6. As you can see, sometimes work needs to be done on the equations before they are ready for substitution.



The key takeaways from this section are as follows:

- Limits help find solutions for functions that may be undefined in certain points.
- For the general limit to exist, the two one-sided limits must exist and must be equal.
- There are ways to find the limit of a function, notably substitution, factoring, and forming the conjugate.

Derivatives

A *derivative* measures the change in a function given a change of one or more of its inputs. In other words, it is the rate of change of a function at a given point.

Having a solid understanding of derivatives is important in building machine learning models, for multiple reasons:

Optimization

To minimize the loss function, optimization methods employ derivatives to ascertain the direction of the steepest descent and modify the model's parameters.

Backpropagation

To execute gradient descent in deep learning, the backpropagation technique uses derivatives to calculate the gradients of the loss function with respect to the model's parameters.

Hyperparameter tuning

To improve the performance of the model, derivatives are used for sensitivity analysis and tuning of hyperparameters.

Do not forget what you learned from the previous section on limits, as you will need this knowledge for this section as well. Calculus mainly deals with derivatives and integrals. This section discusses derivatives and their uses.

You can consider derivatives to be functions that represent (or model) the slope of another function at some point. A *slope* is a measure of a line's position relative to a horizontal line. A positive slope indicates a line moving up, while a negative slope indicates a line moving down.

Derivatives and slopes are related concepts, but they are not the same thing. Here's the main difference between the two:

Slope

The slope measures the steepness of a line. It is the ratio of the change in the y -axis to the change in the x -axis.

Derivative

The derivative describes the rate of change of a given function. As the distance between two points on a function approaches zero, the derivative of that function at that point is the limit of the slope of the tangent line.

Before explaining derivatives in layperson's terms and showing some examples, let's see their formal definitions:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

The equation forms the basis of solving derivatives, although there are many shortcuts that you will learn about. Let's try finding the derivative of a function using the formal definition. Consider the following equation:

$$f(x) = x^2 + 4x - 2$$

To find the derivative, plug $f(x)$ into the formal definition and then solve the limit:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

To simplify things, let's find $f(x + h)$ so that plugging it into the formal definition becomes easier:

$$f(x + h) = (x + h)^2 + 4(x + h) - 2$$

$$f(x + h) = x^2 + 2xh + h^2 + 4x + 4h - 2$$

Now let's plug $f(x + h)$ into the definition:

$$f'(x) = \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 + 4x + 4h - 2 - x^2 - 4x + 2}{h}$$

Notice how there are many terms that can be simplified so that the formula becomes clearer. Remember, you are trying to find the limit for the moment, and the derivative is found after solving the limit:

$$f'(x) = \lim_{h \rightarrow 0} \frac{2xh + h^2 + 4h}{h}$$

The division by h gives further potential for simplification since you can divide all the terms in the numerator by the denominator h :

$$f'(x) = \lim_{h \rightarrow 0} 2x + h + 4$$

It's now time to solve the limit. Because the equation is simple, the first attempt is by substitution, which, as you have guessed, is possible. By substituting the variable h and making it zero (according to the limit), you are left with the following:

$$f'(x) = 2x + 4$$

That is the derivative of the original function $f(x)$. If you want to find the derivative of the function when $x = 2$, you simply have to plug 2 into the derivative function:

$$f'(2) = 2(2) + 4 = 8$$

Figure 4-15 shows the original function's graph with the derivative (the straight line). Notice how $f'(2)$ lies exactly at 8. The slope of $f(x)$ when $x = 2$ is 8.

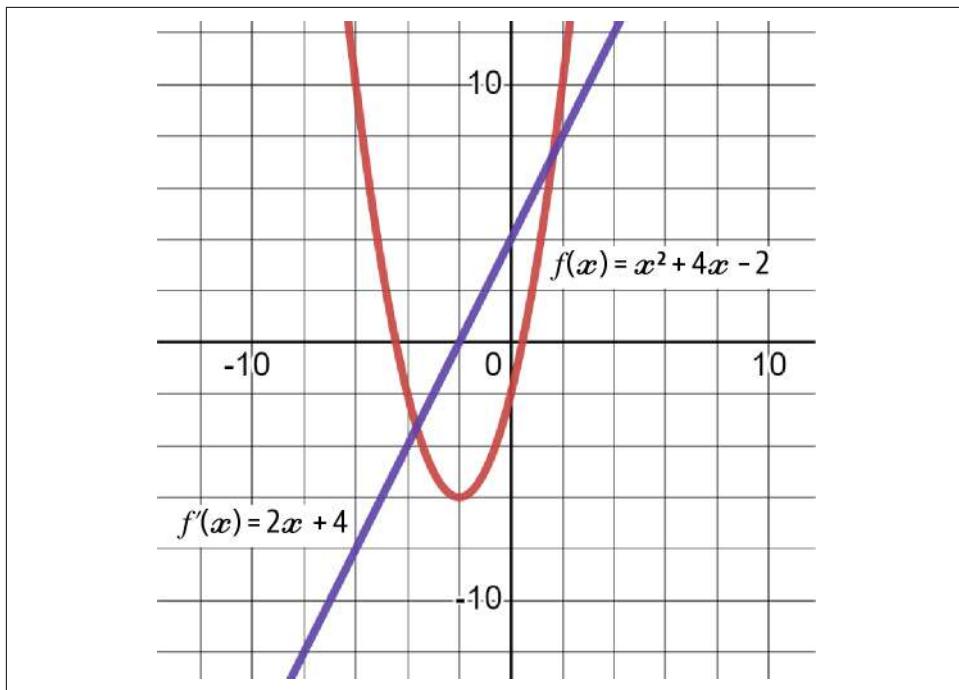


Figure 4-15. The original $f(x)$ with its derivative $f'(x)$



Notice that when $f(x)$ hits the bottom and starts rising, $f'(x)$ crosses the zero line.

You are unlikely to use the formal definition every time you want to find a derivative. There are derivative rules that allow you to save a lot of time through shortcuts. The first rule is referred to as the *power rule*, which is a way to find the derivative of functions with exponents.

It is common to also refer to derivatives using this notation (which is the same thing as $f'(x)$):

$$\frac{dy}{dx}$$

The power rule for finding derivatives is as follows:

$$\frac{dy}{dx}(ax^n) = (a \cdot n)x^{n-1}$$

Basically, this means that the derivative is found by multiplying the constant by the exponent and then subtracting 1 from the exponent. Here's an example:

$$f(x) = x^4$$

$$f'(x) = (1 \times 4)x^{(4 - 1)} = 4x^3$$

Remember that if there is no constant attached to the variable, it means that the constant is equal to 1. Here's a more complex example with the same principle:

$$f(x) = 2x^2 + 3x^7 - 2x^3$$

$$f'(x) = 4x + 21x^6 - 6x^2$$

It is worth noting that the rule also applies to constants even though they do not satisfy the general form of the power rule. The derivative of a constant is zero. While it helps to know why, first you must be aware of the following mathematical concept:

$$x^0 = 1$$

That being said, you can imagine constants as always being multiplied by x to the power of zero (since doing so does not change their value). Now, if you want to find the derivative of 17, here's how it would go:

$$17 = 17x^0 = (0 \times 17)x^{0 - 1} = 0x^{-1} = 0$$

As you know, anything multiplied by zero returns zero as a result. This gives the constants rule for derivatives as follows:

$$\frac{dy}{dx}(a) = 0$$

You follow the same logic when encountering fractions or negative numbers in the exponents.

The *product rule* of derivatives is useful when there are two functions multiplied by each other. The product rule is as follows:

$$\frac{dy}{dx}[f(x)g(x)] = f'(x)g(x) + f(x)g'(x)$$

Let's take an example and find the derivative using the product rule:

$$h(x) = (x^2 + 2)(x^3 + 1)$$

The equation can clearly be segmented into two terms, $f(x)$ and $g(x)$, like this:

$$f(x) = (x^2 + 2)$$

$$g(x) = (x^3 + 1)$$

Let's find the derivatives of the two terms before applying the product rule. Notice that finding the derivative of $f(x)$ and $g(x)$ is easy once you understand the power rule:

$$f'(x) = 2x$$

$$g'(x) = 3x^2$$

When applying the product rule, you should get the following:

$$h'(x) = (x^2 + 2)(3x^2) + (2x)(x^3 + 1)$$

$$h'(x) = 3x^4 + 6x^2 + 2x^4 + 2x$$

$$h'(x) = 5x^4 + 6x^2 + 2x$$

Figure 4-16 shows the graph of $h(x)$ and $h'(x)$.

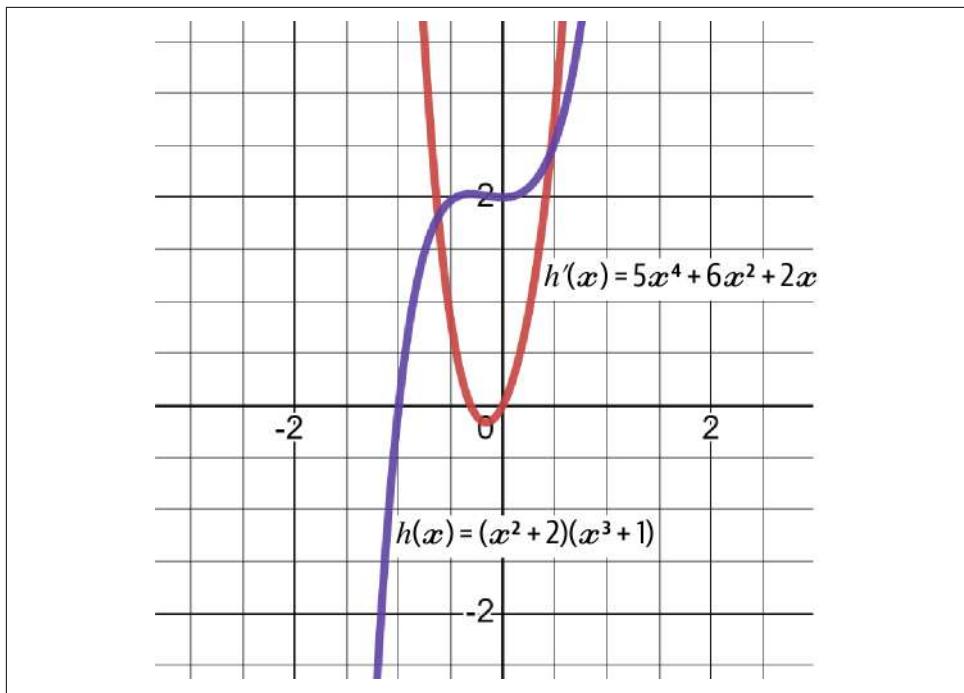


Figure 4-16. The original $h(x)$ with its derivative $h'(x)$

Now let's turn our attention to the *quotient rule*, which deals with the division of two functions. The formal definition is as follows:

$$\frac{dy}{dx} \left[\frac{f(x)}{g(x)} \right] = \frac{f'(x)g(x) - f(x)g'(x)}{\left[g(x) \right]^2}$$

Let's apply it to the following function:

$$f(x) = \frac{x^2 - x + 1}{x^2 + 1}$$

As usual, it's better to start by finding the derivatives of $f(x)$ and $g(x)$, which in this case are clearly separated, with $f(x)$ being the nominator and $g(x)$ being the denominator. When applying the quotient rule, you should get the following:

$$f'(x) = \frac{(2x - 1)(x^2 + 1) - (x^2 - x + 1)(2x)}{(x^2 + 1)^2}$$

$$f'(x) = \frac{2x^3 + 2x - x^2 - 1 - 2x^3 + 2x^2 - 2x}{(x^2 + 1)^2}$$

$$f'(x) = \frac{x^2 - 1}{(x^2 + 1)^2}$$

Exponential derivatives deal with the power rule applied to constants. Take a look at the following equation. How would you find its derivative?

$$f(x) = a^x$$

Instead of the usual variable-base-constant-exponent, it is constant-base-variable-exponent. This is treated differently when trying to calculate the derivative. The formal definition is as follows:

$$\frac{dy}{dx} a^x = a^x (\ln a)$$

The following example shows how this is done:

$$\frac{dy}{dx} 4^x = 4^x (\ln 4)$$

Euler's number, mentioned earlier, has a special derivative. When it comes to finding the derivative of e , the answer is interesting:

$$\frac{dy}{dx} e^x = e^x (\ln e) = e^x$$

This is because the natural log function and the exponential function are inverses of each other, so the term $\ln e$ equals 1. Therefore, the derivative of the exponential function e is itself.

In parallel, let's discuss logarithmic derivatives. By now, you should know what exponents and logarithms are. The general definition for both types of logarithms is as follows:

$$\frac{dy}{dx} \log_a x = \frac{1}{x \ln a}$$

$$\frac{dy}{dx} \ln x = \log_e x = \frac{1}{x \ln e} = \frac{1}{x}$$

Notice how in the second derivative function of the natural logarithm, the term $\ln e$ is once again encountered, thus making simplification quite easy since it is equal to 1.

Take the following example:

$$f(x) = 7\log_2(x)$$

Using the formal definition, the derivative of this logarithmic function is as follows:

$$f'(x) = 7\left(\frac{1}{x \ln 2}\right) = \frac{7}{x \ln 2}$$



The logarithm \log has a base of 10, but the natural logarithm \ln has a base of e (~2.7182).

The natural logarithm and the log function are actually linearly related through simple multiplication. If you know the log of the constant a , you can find its natural logarithm \ln by multiplying the log of a by 2.304.

One major concept in derivatives is the *chain rule*. Let's back up to the power rule, which deals with exponents on variables. Remember the following formula to find the derivative:

$$\frac{dy}{dx}(ax^n) = (a \cdot n)x^{n-1}$$

This is a simplified version because there is only x , but the reality is that you must multiply by the derivative of the term under the exponent. Until now, you have seen only x as the variable under the exponent. The derivative of x is 1, which is why it is simplified and rendered invisible. However, with more complex functions such as this one:

$$f(x) = (4x + 1)^2$$

The derivative of the function is found by following these two steps:

1. Find the derivative of the outside function without touching the inside function.
2. Find the derivative of the inside function and multiply it by the rest of the function.

The solution is therefore as follows (knowing that the derivative of $4x + 1$ is just 4):

$$f'(x) = 2(4x + 1).4$$

$$f'(x) = 8(4x + 1)$$

$$f'(x) = 32x + 8$$

The same applies with the exponential functions. Take the following example:

$$f(x) = e^x$$

$$f'(x) = e^x(1) = e^x$$

The chain rule can actually be considered a master rule as it applies anywhere, even in the product rule and the quotient rule.

There are more concepts to master in derivatives, but as this book is not meant to be a full calculus master class, you should at least know the meaning of a derivative, how it is found, what it represents, and how it can be used in machine and deep learning.



The key takeaways from this section are as follows:

- A derivative measures the change in a function given a change of one or more of its inputs.
- The power rule is used to find the derivative of a function raised to a power.
- The product rule is used to find the derivative of two functions that are multiplied together.
- The quotient rule is used to find the derivative of two functions that are divided by each other.
- The chain rule is the main rule used in differentiating (which means the process of finding the derivative). Due to simplicity, it is often overlooked.
- Derivatives play a crucial role in machine learning, such as enabling optimization techniques, aiding model training, and enhancing the interpretability of the models.

Integrals and the Fundamental Theorem of Calculus

An *integral* is an operation that represents the area under a curve of a function given an interval. It is the inverse of a derivative, which is why it is also called an *antiderivative*.

The process of finding integrals is called *integration*. Integrals can be used to find areas below a curve, and they are heavily used in the world of finance in such areas as risk management, portfolio management, probabilistic methods, and even option pricing.

The easiest way to understand an integral is to think of calculating an area below the curve of a function. This can be done by manually calculating the different changes in the x -axis, but adding these slices to find the area is a tedious process. This is where integrals come to the rescue.

Keep in mind that an integral is the inverse of a derivative. This is important because it implies a direct relationship between the two. The basic definition of an integral is as follows:

$$\int f(x) dx = F(X) + C$$

The \int symbol represents the integration process

$f(x)$ is the derivative of the general function $F(x)$

C represents the lost constant in the differentiation process

dx represents slicing along x as it approaches zero

The preceding equation means that the integral of $f(x)$ is the general function $F(x)$ plus a constant C , which was lost in the initial differentiation process. Here's an example to better explain the need to put in the constant.

Consider the following function:

$$f(x) = x^2 + 5$$

Calculating its derivative, you get the following result:

$$f'(x) = 2x$$

Now, what if you wanted to integrate it so that you go back to the original function (which in this case is represented by the capital letter $F(x)$ instead of $f(x)$)?

$$\int 2x dx$$

Normally, having seen the differentiation process (which means taking the derivative), you would return 2 as the exponent, which gives you the following answer:

$$\int 2x \, dx = x^2$$

This does not look like the original function. It's missing the constant 5. But you have no way of knowing that, and even if you knew there was a constant, you would have no way of knowing what it is: 1? 2? 677? This is why a constant C is added in the integration process to represent the lost constant. Therefore, the answer to the integration problem is as follows:

$$\int 2x \, dx = x^2 + C$$



Up until now, the discussion has been limited to *indefinite integrals* where the integration symbol is *naked* (which means there are no boundaries to it). You will see what this means right after we define the necessary rules to complete the integration.

For the power function (just like the previous function), the general rule for integration is as follows:

$$\int x^a \, dx = \frac{x^{a+1}}{a+1} + C$$

This is much simpler than it looks. You are just reversing the power rule you saw earlier. Consider the following example:

$$\int 2x^6 \, dx$$

$$\int 2x^6 \, dx = \frac{2x^7}{7} + C$$

$$\int 2x^6 \, dx = \frac{2}{7}x^7 + C$$

To verify your answer, you can find the derivative of the result (using the power rule):

$$F(x) = \frac{2}{7}x^7 + C$$

$$f'(x) = (7)\frac{2}{7}x^{7-1} + 0$$

$$f'(x) = 2x^6$$

Let's take another example. Consider the following integration problem:

$$\int 2 \, dx$$

Naturally, using the rule, you should find the following result:

$$\int 2 \, dx = 2x + C$$

Let's move on to *definite integrals*, which are integrals with numbers on the top and bottom that represent intervals below a curve of a function. Hence, *indefinite* integrals find the area under the curve everywhere, and definite integrals are bounded within an interval given by point a and point b . The general definition of indefinite integrals is as follows:

$$\int_a^b f(x) \, dx = F(B) - F(A)$$

This is as simple as it gets. You will solve the integral, then plug in the two numbers and subtract the two functions from each other. Consider the following evaluation of an integral (integral solving is commonly referred to as *evaluating* the integral):

$$\int_0^6 3x^2 - 10x + 4 \, dx$$

The first step is to understand what is being asked. From the definition of integrals, it seems that the area between $[0, 2]$ on the x -axis is to be calculated using the given function:

$$F(x) = ([x^3 - 5x^2 + 4x + C])|_0^6$$

To evaluate the integral at the given points, simply plug in the values as follows:

$$F(x) = ([6^3 - 5(6)^2 + 4(6) + C]) - ([0^3 - 5(0)^2 + 4(0) + C])$$

$$F(x) = ([216 - 180 + 24 + C]) - ([0 - 0 + 0 + C])$$

$$F(x) = ([60 + C]) - ([0 + C])$$

$$F(x) = (60 - 0)$$

$$F(x) = 60$$



The constant C will always cancel out indefinite integrals, so you can leave it out in this kind of problem.

Therefore, the area below the graph of $f(x)$ and above the x -axis, as well as between $[0, 6]$ on the x -axis, is equal to 60 square units. The following shows a few rules of thumb on integrals (after all, this chapter is supposed to refresh your knowledge or give you a basic understanding of a few key mathematical concepts):

- To find the integral of a constant:

$$\int a \, dx = ax + C$$

- To find the integral of a variable:

$$\int x \, dx = \frac{1}{2}x^2 + C$$

- To find the integral of a reciprocal:

$$\int \frac{1}{x} \, dx = \ln |x| + C$$

- To find the integral of an exponential:

$$\int a^x \, dx = \frac{a^x}{\ln(a)} + C$$

$$\int e^x \, dx = e^x + C$$

The *fundamental theorem of calculus* links derivatives with integrals. This means that it defines derivatives in terms of integrals and vice versa. The fundamental theorem of calculus is actually made up of two parts:

Part I

The first part of the fundamental theorem of calculus states that if you have a continuous function $f(x)$, then the original function $F(x)$ defined as the antiderivative of $f(x)$ from a fixed starting point a up to x is a function that is differentiable everywhere from a to x , and its derivative is simply $f(x)$ evaluated at x .

Part II

The second part of the fundamental theorem of calculus states that if you have a function $f(x)$ that is continuous over a certain interval $[a, b]$, and you define a new function $F(x)$ as the integral of $f(x)$ from a to x , then the definite integral of $f(x)$ over that same interval $[a, b]$ can be calculated as $F(b) - F(a)$.

The theorem is useful in many fields, including physics and engineering, but optimization and other mathematical models also benefit from it. Some examples of using integrals in the different learning algorithms can be summed up as follows:

Density estimation

Integrals are used in density estimation, a part of many machine learning algorithms, to calculate the probability density function.

Reinforcement learning

Integrals are used in reinforcement learning to calculate expected values of reward functions. Reinforcement learning is covered in [Chapter 10](#).



The key takeaways from this section are as follows:

- Integrals are also known as antiderivatives and they are the opposite of derivatives.
- Indefinite integrals find the area under the curve everywhere, while definite integrals are bounded within an interval given by point a and point b .
- The fundamental theorem of calculus is the bridge between derivatives and integrals.
- In machine learning integrals are used for modeling uncertainty, making predictions, and estimating expected values.

Optimization

Several machine and deep learning algorithms depend on optimization techniques to decrease error functions.

Optimization is the process of finding the best solution among all possible solutions. Optimization is all about finding the highest and lowest points of a function. [Figure 4-17](#) shows the graph for the following formula:

$$f(x) = x^4 - 2x^2 + x$$

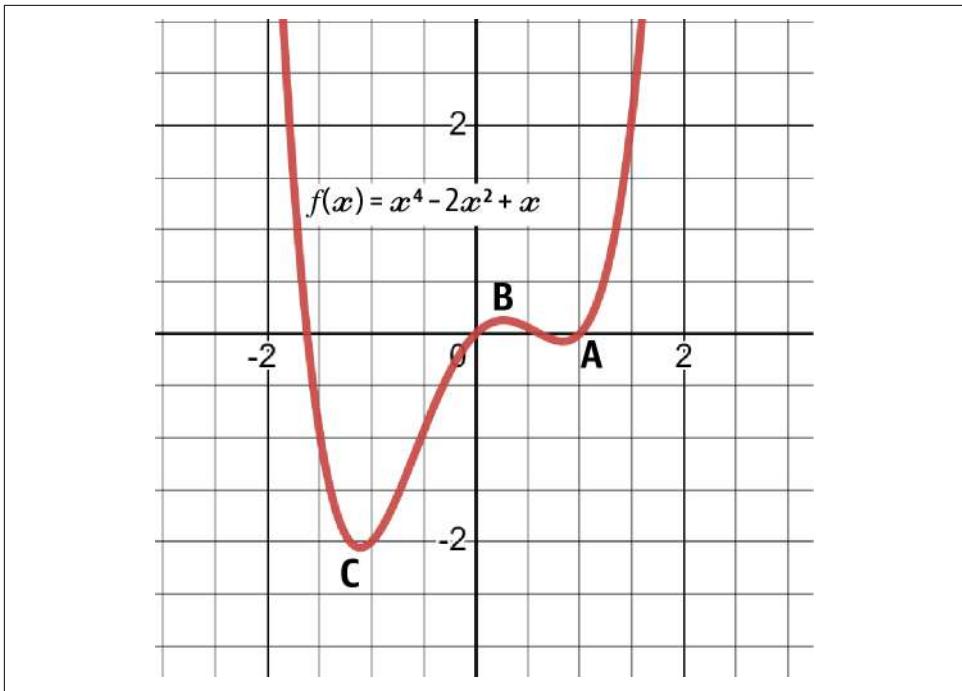


Figure 4-17. A graph of the function $f(x) = x^4 - 2x^2 + x$

A *local minimum* exists when values on the right of the x -axis are decreasing until reaching a point where they start increasing. The point does not have to necessarily be the lowest point in the function, hence the name *local*. In Figure 4-17, the function has a local minimum at point A.

A *local maximum* exists when values on the right of the x -axis are increasing until reaching a point where they start decreasing. The point does not have to necessarily be the highest point in the function. In Figure 4-17, the function has a local maximum at point B.

A *global minimum* exists when values on the right of the x -axis are decreasing until reaching a point where they start increasing. The point must be the lowest point in the function, hence the name *global*. In Figure 4-17, the function has a global minimum at point C.

A *global maximum* exists when values on the right of the x -axis are increasing until reaching a point where they start decreasing. The point must be the highest point in the function. In [Figure 4-17](#), there is no global maximum, as the function will continue infinitely without creating a top. You can clearly see how the function accelerates upward.

When dealing with machine and deep learning models, the aim is to find model parameters (or inputs) that minimize what is known as a *loss function* (a function that gives the error of forecasts). If the loss function is convex, optimization techniques should find the parameters that tend toward the global minimum where the loss function is minimized.

If the loss function is nonconvex, the convergence is not guaranteed, and the optimization may only lead toward approaching a local minimum, which is a part of the aim, but this leaves the global minimum, which is the final aim.

But how are these minima and maxima found? Let's look at it step by step:

1. The first step is to perform the first derivative test (which is calculating the derivative of the function). Then, setting the function equal to zero and solving for x will give what are known as critical points. *Critical points* are the points where the function changes direction (the values stop going in one direction and start going in another direction). Therefore, these points are maxima and minima.
2. The second step is to perform the second derivative test (which is simply calculating the derivative of the derivative). Then, setting the function equal to zero and solving for x will give what are known as inflection points. *Inflection points* show where the function is concave up and where it is concave down.

In other words, critical points are where the function changes direction, and inflection points are where the function changes concavity. [Figure 4-18](#) shows the difference between a concave up function and a concave down function.

$$\text{Concave up function} = x^2$$

$$\text{Concave down function} = -x^2$$

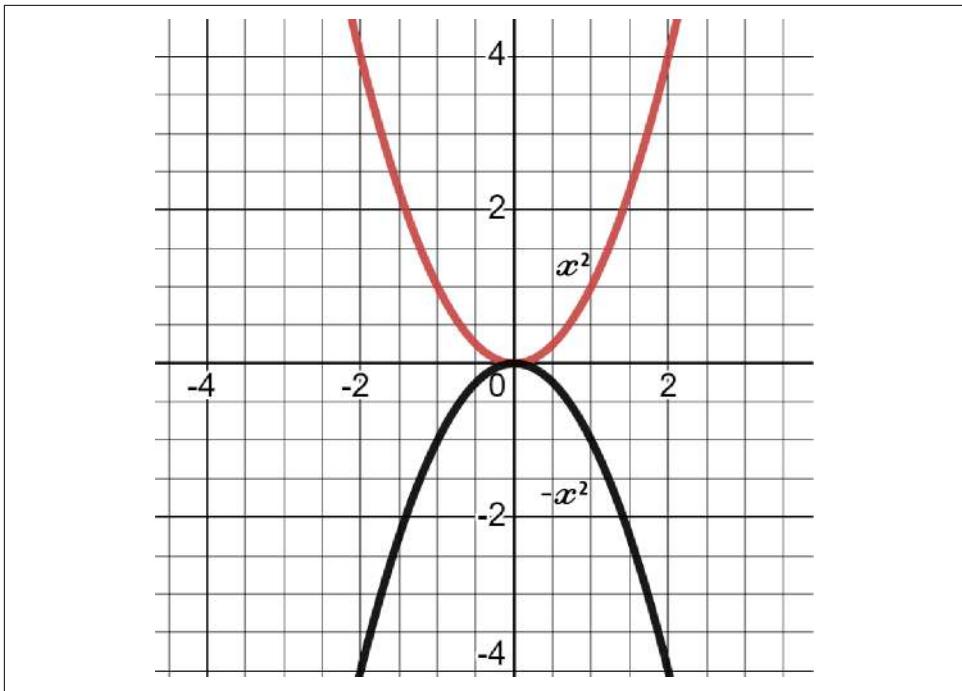


Figure 4-18. A concave up function and a concave down function

The steps to find the extrema are as follows:

1. Find the first derivative and set it to zero.
2. Solve the first derivative to find x . The values are called critical points, and they represent the points where the function changes direction.
3. Plug values into the formula that are either below or above the critical points. If the result of the first derivative is positive, it means that it's increasing around that point, and if it's negative, it means that it's decreasing around that point.
4. Find the second derivative and set it to zero.
5. Solve the second derivative to find x . The values, called inflection points, represent the points where concavity changes from up to down and vice versa.
6. Plug values into the formula that are either below or above the inflection points. If the result of the second derivative is positive, it means there is a minimum at that point, and if it's negative, it means there is a maximum at that point.

It is important to understand that the first derivative test relates to critical points and the second derivative test relates to inflection points. The following example finds the extrema of the function:

$$f(x) = x^2 + x + 4$$

The first step is to take the first derivative, set it to zero, and solve for x :

$$f'(x) = 2x + 1$$

$$2x + 1 = 0$$

$$x = -\frac{1}{2}$$

The result shows there is a critical point at that value. Now find the second derivative:

$$f''(x) = 2$$

Next, the critical point must be plugged into the second derivative formula:

$$f''\left(-\frac{1}{2}\right) = 2$$

The second derivative is positive at the critical point. This means that there is a local minimum at that point.

In the coming chapters, you will see more complex optimization techniques such as the gradient descent and the stochastic gradient descent, which are fairly common in machine learning algorithms. Note that you do not have to fully understand the details of optimization and solving for the unknown variables as the algorithms will do that on their own.



The key takeaways from this section are as follows:

- Optimization is the process of finding the function's extrema.
- Critical points are the points where the function changes direction.
- Inflection points give where the function is concave up and where it is concave down.
- A loss function is a function that measures the error of forecasts in predictive machine learning.

Summary

Chapters 2, 3, and 4 presented the main numerical concepts to help you start understanding basic machine and deep learning models. I made all reasonable efforts to simplify the technical details as much as possible. However, I encourage you to read these three chapters at least twice so that everything you have learned becomes second nature. I also encourage you to research these concepts in more depth in other material.

Naturally, deep learning requires more in-depth knowledge in mathematics, but I believe that with the concepts in this chapter, you may start dipping your toes into creating algorithms. After all, they come prebuilt from packages and libraries, and the aim of this chapter was to help you understand what you are working with. It is unlikely that you will build the models from scratch using archaic tools.

By now, you should have gained a certain understanding of data science and the mathematical requirements that will get you started comfortably. We have two more topics to cover before you can start building your first machine learning model: technical analysis and Python for data science.

Introducing Technical Analysis

Technical analysis presents many types of inputs (explanatory variables) that you can use in your deep learning models. This chapter introduces this vast field so that you are equipped with the necessary knowledge to create technical-based learning models in the chapters to follow.

Technical analysis in finance relies on the visual interpretation of a price action's history to determine the likely aggregate direction of the market. It relies on the idea that the past is the best predictor of the future. There are several types of techniques within the vast field that is technical analysis, notably the following:

Charting analysis

This is where you apply subjective visual interpretation techniques onto charts. You generally use methods like drawing support and resistance lines as well as retracements to find inflection levels that aim to determine the next move.

Indicator analysis

This is where you use mathematical formulas to create objective indicators that can be either trend following or contrarian. Among known indicators are *moving averages* and the *relative strength index* (RSI), both of which are discussed in greater detail in this chapter.

Pattern recognition

This is where you monitor certain recurring configurations and act on them. A *pattern* is generally an event that emerges from time to time and presents a certain theoretical or empirical outcome. In finance, it is more complicated, but certain patterns have been shown to add value across time, and this may partly be due to a phenomenon called *self-fulfilling prophecy* (a process by which an initial expectation leads to its confirmation).

Let's take a quick tour of the history of technical analysis so that you have a better idea of what to expect. Technical analysis relies on three principles:

History repeats itself

You are likely to see clusters during trends and ranges. Also, certain configurations are likely to have a similar outcome most of the time.

The market discounts everything

It is assumed that everything (all fundamental, technical, and quantitative information) is included in the current price.

The market moves in waves

Due to different time frames and needs, traders buy and sell at different frequencies, therefore creating trends and waves as opposed to a straight line.

Unfortunately, technical analysis is overhyped and misused by the retail trading community, which gives it a somewhat less-than-savory reputation in the professional industry. Every type of analysis has its strengths and weaknesses, and there are successful fundamental, technical, and quantitative investors, but there are also failed investors from the three fields.

Fundamental analysis relies on economic and financial data to deliver a judgment on a specific security or currency with a long-term investment horizon, whereas *quantitative analysis* is more versatile and is more often applied to short-term data. It uses mathematical and statistical concepts to deliver a forecast or to manage risk.

Among other assumptions, technical analysis suggests that markets are not efficient, but what does that mean? *Market efficiency* states that information is already embedded in the current price and that price and value are the same thing. When you buy an asset, you are hoping that it is *undervalued* (in fundamental analysis jargon) or *oversold* (in technical analysis jargon), which is why you believe the price should go up to meet the value. Therefore, you are assuming that the value is greater than the price.

Market efficiency rebuffs any claims that the price does not equal the value and therefore suggests that any alpha trading must not result in above-average returns (*alpha trading* is the act of engaging in speculative operations to perform better than a benchmark, which is typically an index).

The market efficiency hypothesis is the technical analyst's greatest enemy, as one of its principles is that in the weak form of efficiency, you cannot earn excess returns from technical analysis. Hence, technical analysis gets shot down right at the beginning, and then fundamental analysis gets its share of the beating.

It is fair to assume that at some point in the future, markets will have no choice but to be efficient due to the number of participants and the ease of access to information. However, as political and abnormal events show us, markets tend to be anything but efficient.



An example of a political event that triggered panic and irrationality in the markets is the Russia-Ukraine war that started in 2022. An example of an abnormal economic event is an unexpected interest rate hike from a central bank.

Charting Analysis

Before you can understand what charting analysis is, you need to know what you see when opening a chart—or more specifically, a candlestick chart.

Let's assume that the market for a particular stock opens at \$100. Some trading activity occurs. Let's also record the high price (\$102) and the low price (\$98) printed during the hourly period. Also, record the hourly close price (\$101). Recall that these four pieces of data are referred to as *open*, *high*, *low*, and *close* (OHLC). They represent the four basic prices that are necessary to create candlestick charts.

Candlesticks are extremely simple and intuitive. They are box-shaped chronological elements across the timeline that contain the OHLC data. [Figure 5-1](#) shows everything you need to know about how a candlestick works.

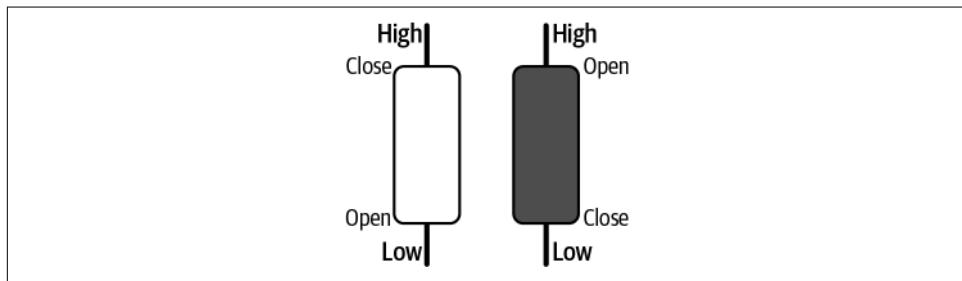


Figure 5-1. On the left, a bullish candlestick; on the right, a bearish candlestick

A *bullish* candlestick has a close price higher than its open price, whereas a *bearish* candlestick has a close price lower than its open price.

Candlestick charts are among the most common ways to analyze financial time series. They contain more information than simple line charts and offer more visual interpretability than bar charts.



A *line chart* is created by joining the close prices chronologically. It is the simplest way to chart an asset. It contains the least information among the three chart types since it shows only the close price.

Charting analysis is the task of finding support and resistance lines through subjective drawing. *Lines*, whether horizontal or diagonal, are the essence of finding the following levels to predict the market's reaction:

Support level

A level from where the market should bounce, as it is implied that demand should be higher than the supply around it

Resistance level

A level from where the market should retreat, as it is implied that supply should be higher than the demand around it

The asset's direction on a timeline axis can be threefold: *uptrend*, where prices are making higher highs; *downtrend*, where prices are making lower lows; and *sideways* (or ranging), where prices fluctuate around the same level for extended periods of time.

Figure 5-2 shows a horizontal support level on the EURUSD (the value of one euro priced in US dollars) close to 1.0840. Generally, traders start thinking about buying when a price is close to support. This is in anticipation of a reaction to the upside since the balance of power should shift more to the demand (positive) side, where traders accept paying a higher price as they expect an even higher price in the future (remember the price-to-value argument discussed earlier). The implication here is that most traders see a price that is lower than the value.

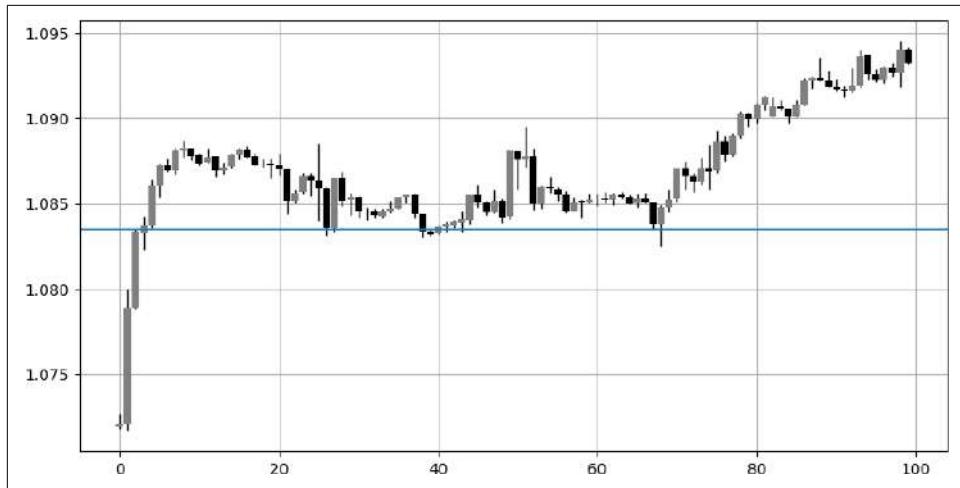


Figure 5-2. Candlestick chart on the EURUSD showing support at 1.0840

Figure 5-3 shows a resistance level on the USDCHF (the value of one US dollar priced in Swiss francs) close to 0.9030. Generally, traders start thinking about shorting the market when it is close to resistance. This is in anticipation that a reaction to the downside should occur since the balance of power should shift more to the supply side. The implication here is that most traders see a price that is higher than the value.

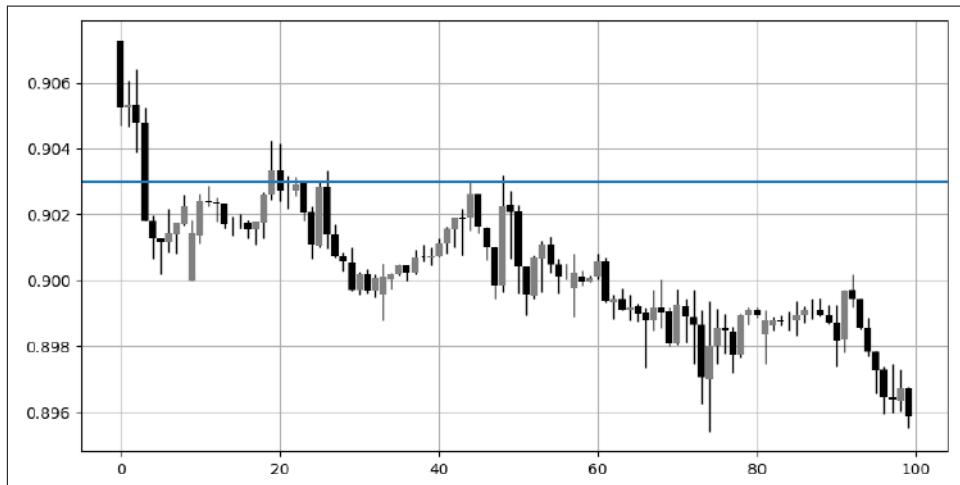


Figure 5-3. Candlestick chart on the USDCHF showing resistance at 0.9030

Ranging (sideways) markets give more confidence that horizontal support and resistance lines will work. This is because of the already implied general balance between supply and demand. Therefore, if there is excess supply, the market would adjust quickly, as demand should rise enough to stabilize the price.

Figure 5-4 shows a ranging market trapped between two horizontal levels; this is the case of the USDCAD (the value of one US dollar priced in Canadian dollars). Whenever the market approaches the resistance line in a ranging market, you should have more confidence that a drop will occur than you would in a rising market, and whenever it approaches support, you should have more confidence that a bounce will occur than you would in a falling market.

Charting analysis is also applied on trending markets. This comes in the form of ascending and descending channels. They share the same inclination as horizontal levels but with a bias (discussed later).

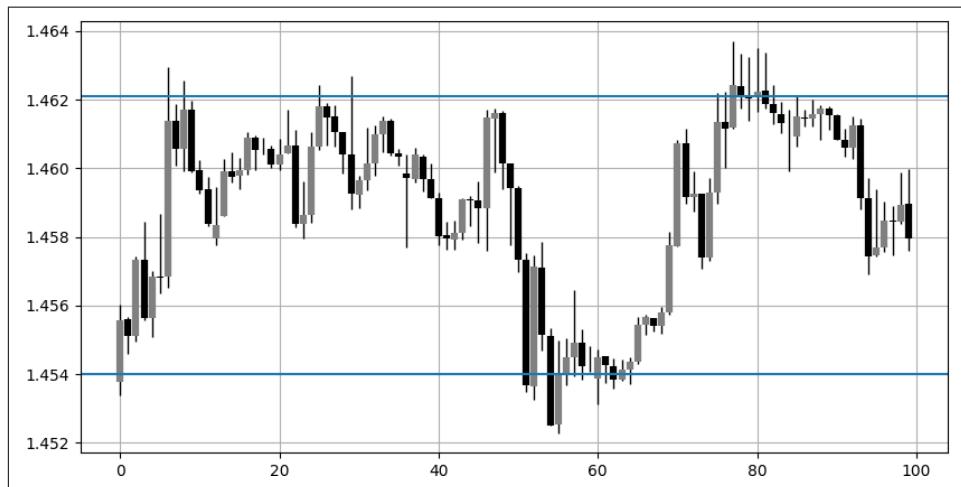


Figure 5-4. Candlestick chart on the USDCAD showing support at 1.4540 and resistance at 1.4620

Figure 5-5 shows an *ascending channel* where support and resistance points rise over time to reflect the bullish pressure stemming from a steadily rising demand force.

Traders seeing this would anticipate a bullish reaction whenever the market approaches the lower part of the ascending channel and would expect a bearish reaction whenever the market approaches the upper part of the channel.

This has no sound scientific backing, because nothing says that the market must move in parallel, but the self-fulfilling prophecy may be why such channels are considered predictive in nature.



Figure 5-5. Candlestick chart on the EURUSD showing an ascending channel

Figure 5-6 shows a descending channel where support and resistance points fall with time to reflect the bearish pressure coming from a steadily rising supply force. Generally, bearish channels tend to be more aggressive as fear dominates greed, and sellers are more panicky than buyers are greedy.

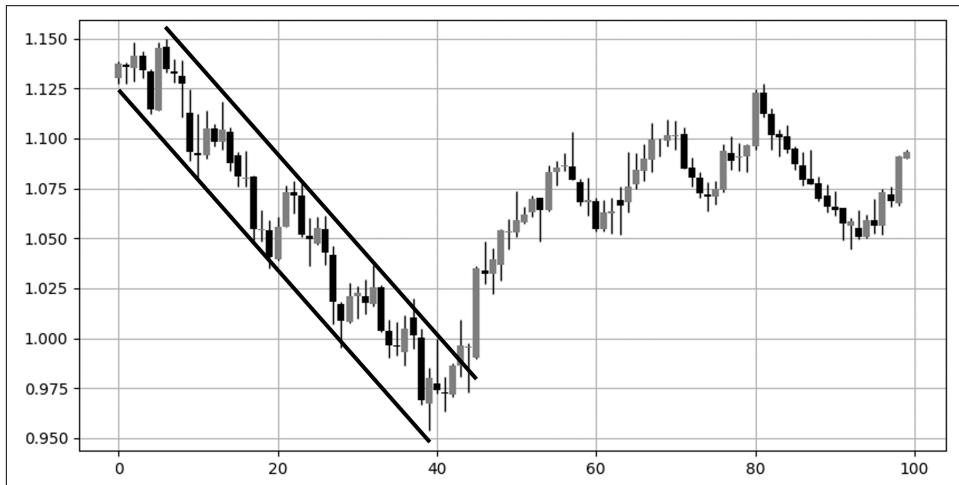


Figure 5-6. Candlestick chart on the EURUSD showing a descending channel

I mentioned a bias when dealing with ascending and descending channels. I refer to this bias as the *invisible hand*. Here's why:

"The trend is your friend." This saying, coined by stock investor, investment adviser, and financial analyst Martin Zweig, means that with ascending channels, you need to be focusing more on buying whenever the market reverts to the support zone. That's because you want the invisible hand of the bullish pressure to increase your probability of a winning trade. Similarly, in the case of a descending channel, you should focus more on short selling whenever the market reaches the upper limit. The full version of Zweig's axiom goes as follows: "The trend is your friend, until the end when it bends." This means that at any point in time, the market may change its regime, and any friendship with the trend gets terminated. In the end, charting analysis is subjective in nature and relies more on the experience of the trader or analyst.

It is worth mentioning that there are many ways to find support and resistance levels other than drawing them through visual estimation:

Fibonacci retracements

This is where you use Fibonacci ratios to give out reactionary levels. Fibonacci retracements are usually calculated on up or down legs so that you know where the market will reverse if it touches one of these levels. The problem with this method is that it is very subjective and, as with any other technique, not perfect. The advantage is that it gives many interesting levels.

Pivot points

With pivot points, you use simple mathematical formulas to find levels. Based on yesterday's trading activity, you use formulas to project today's future support and resistance levels. Then, whenever the market approaches the levels, you try to fade the move by trading in the opposite direction.

Moving averages

These are discussed in the next section. They are dynamic in nature and follow the price. You can also use them to detect the current market regime.



The best way to find support and resistance levels is to combine as many techniques as possible so that you have a certain confluence of methods, which in turn will increase your conviction for the initial idea. Trading is a numbers game, and stacking the odds on your side as much as possible should eventually increase your chances for a better-performing system.

Indicator Analysis

Indicator analysis is the second-most used technical analysis tool. It generally accompanies charting to confirm your initial idea. You can think of *indicators* as assistants. They can be divided into two types:

Trend-following indicators

Used to detect and trade a trending market where the current move is expected to continue. Therefore, they are related to the persistence of the move.

Contrarian indicators

Used to fade the move¹ and are best used in sideways markets² as they generally signal the end of the initial move. Therefore, they are related to the expected reversal of the move (and therefore to the antipersistence of the move).

The next sections present two pillars of technical analysis: moving averages (trend following) and the relative strength index (contrarian).



Indicators are important, as you will use them as inputs in the different learning algorithms in subsequent chapters.

Moving Averages

The most famous trend-following overlay indicator is the *moving average*. Its simplicity makes it without a doubt one of the most sought-after tools. Moving averages help confirm and ride the trend. You can also use them to find support and resistance levels, stops, and targets, as well as to understand the underlying trend.

There are many types of moving averages, but the most common is the simple moving average where you take a rolling mean of the close price, as shown in the following formula:

$$\text{Moving average}_i = \frac{\text{Price}_i + \text{Price}_{i-1} + \dots + \text{Price}_{i-n}}{n}$$

¹ Fading the move is a trading technique where you trade in the opposite direction of the ongoing trend in the hope that you are able to time its end.

² Sideways markets are generally in equilibrium and no specific trend describes them. They tend to swing from tops to bottoms that are close to each other.

Figure 5-7 shows the 30-hour simple moving average applied on the USDCAD. The term *30-hour* means that I calculate the moving average of the latest 30 periods in case of hourly bars.

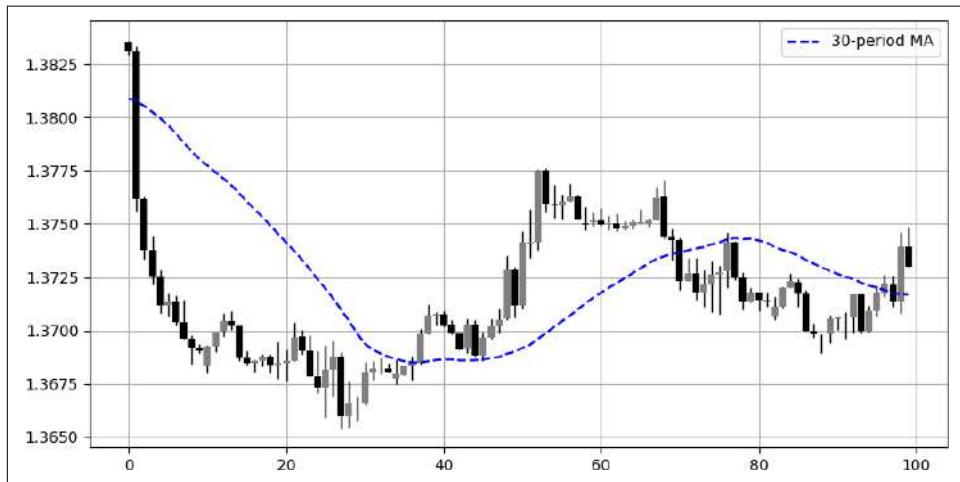


Figure 5-7. Candlestick chart on the USDCAD with a 30-hour simple moving average

Rules of thumb with moving averages include the following:

- Whenever the market is above its moving average, a bullish momentum is in progress, and you are better off looking for long opportunities.
- Whenever the market is below its moving average, a bearish momentum is in progress, and you are better off looking for short opportunities.
- Whenever the market crosses over or under its moving average, you can say that the momentum has changed and that the market may be entering a new regime (trend).

You can also combine moving averages so that they give out signals. For example, whenever a short-term moving average crosses over a long-term moving average, a bullish crossover has occurred, and the market may continue to rise. This is also referred to as a *golden cross*.

In contrast, whenever a short-term moving average crosses under a long-term moving average, a bearish crossover has occurred, and the market may continue to drop. This is also referred to as a *death cross*.

Figure 5-8 shows the USDCAD with a 10-hour (closer to the market price) and a 30-hour moving average (farther from the market price).

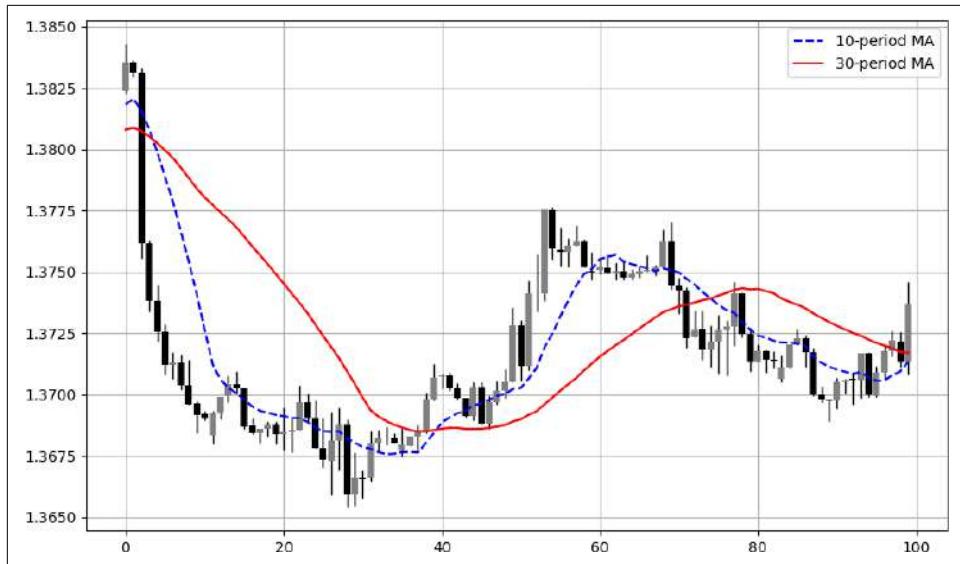


Figure 5-8. Candlestick chart on the USDCAD with a 30-hour and a 10-hour simple moving average

The Relative Strength Index

Let's now look at the contrarian indicator. First introduced by J. Welles Wilder Jr.,³ the *relative strength index* (RSI) is one of the most popular and versatile bounded indicators. It is mainly used as a contrarian indicator where extreme values signal a reaction that can be exploited. Use the following steps to calculate the default 14-period RSI:

1. Calculate the change in the closing prices from the previous ones.
2. Separate the positive net changes from the negative net changes.
3. Calculate a smoothed moving average on the positive net changes and on the absolute values of the negative net changes.
4. Divide the smoothed positive changes by the smoothed absolute negative changes. Refer to this calculation as the *relative strength* (RS).
5. Apply this normalization formula for every time step to get the RSI:

$$RSI_i = 100 - \frac{100}{1 + RS_i}$$

³ See *New Concepts in Technical Trading Systems* by J. Welles Wilder Jr. (Trend Research).



The *smoothed* moving average is a special type of moving average developed by the creator of the RSI. It is smoother and more stable than the simple moving average.

Generally, the RSI uses a lookback period of 14 by default, although each trader may have their own preferences on this. Here's how to use this indicator:

- Whenever the RSI is showing a reading of 30 or less, the market is considered to be oversold, and a correction to the upside might occur.
- Whenever the RSI is showing a reading of 70 or more, the market is considered to be overbought, and a correction to the downside might occur.
- Whenever the RSI surpasses or breaks the 50 level, a new trend might be emerging, but this is generally a weak assumption and more theoretical than practical in nature.

Figure 5-9 shows the EURUSD versus its 14-period RSI in the second panel. Indicators should be used to confirm long or short bias and are very helpful in timing and analyzing the current market state.

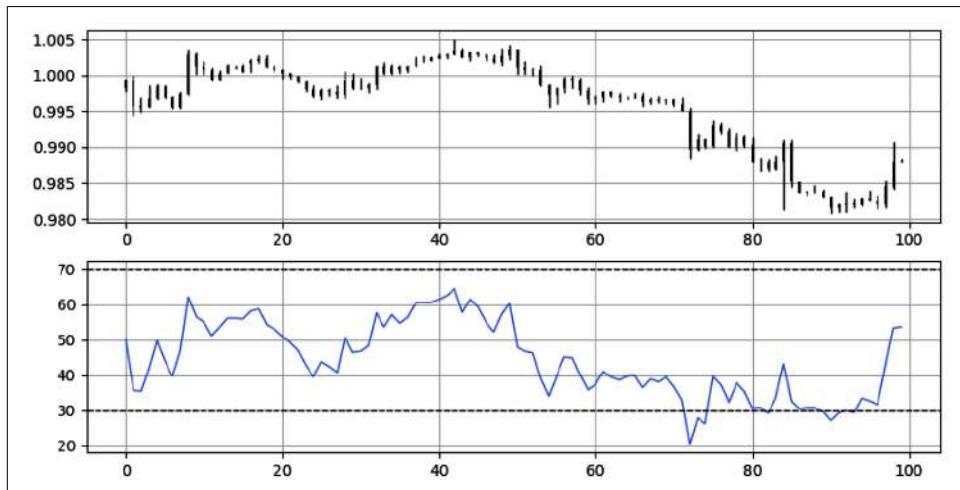


Figure 5-9. Hourly EURUSD values in the top panel with the 14-period RSI in the bottom panel

To summarize, indicators can be calculated in many ways. The two most commonly used ones are moving averages and the RSI.

Pattern Recognition

Patterns are recurring configurations that show a specific prediction of the ensuing move. Patterns can be divided into the following types:

Classic price patterns

These are known as technical reversal price patterns, which are extremely subjective and can be considered unreliable due to the difficulty of backtesting them without taking subjective conditions. However, they are still used by many traders and analysts.

Timing patterns

Based on a combination of timing and price, these patterns are less well known but can be powerful and predictive when used correctly.

Candlestick patterns⁴

This is where OHLC data is used to predict the future reaction of the market. Candlesticks are one of the best ways to visualize a chart as they harbor many patterns that could signal reversals or confirm the move.

Classic price patterns refer to theoretical configurations such as double tops and rectangles. They are usually either reversal or continuation patterns:

Continuation price patterns

These are configurations that confirm the aggregate ongoing move. Examples include rectangles and triangles.

Reversal price patterns

These are configurations that fade the aggregate ongoing move. Examples include head and shoulders and double bottoms.

Old-school chartists are familiar with double tops and bottoms, which signal reversals and give the potential of said reversals. Despite their simplicity, they are subjective, and some are not visible like others.

This hinders the ability to know whether they add value or not. [Figure 5-10](#) shows an illustration of a double top where a bearish bias is given right after the validation of the pattern, which is usually breaking the line linking the lows of the bottom between the two tops. This line is called the *neckline*.

⁴ See my book [Mastering Financial Pattern Recognition](#) (O'Reilly) for a more in-depth discussion on candlestick patterns.

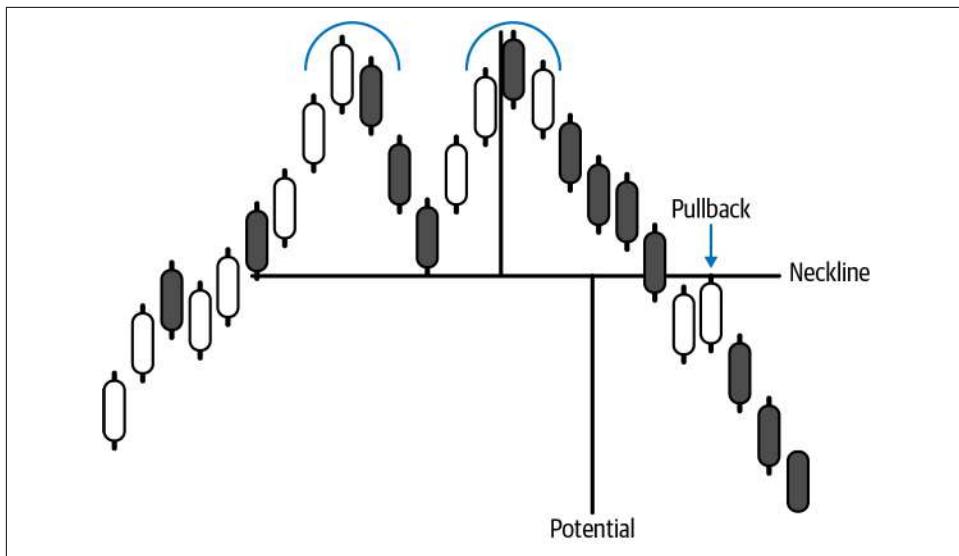


Figure 5-10. Double top illustration

Notice these three important elements in a double top:

The neckline

This is the line linking the lowest low between the two peaks and the beginning/end of the pattern. It serves to determine the pullback level.

The pullback

Having broken the neckline, the market should shape a desperate attempt toward the neckline but fails to continue higher as the sellers use this level as a reentry to continue shorting. Therefore, the pullback level is the theoretical optimal selling point after validating a double top.

The potential

This is the target of the double top. It is measured as the midpoint between the top of the pattern and the neckline projected lower and starting from the same neckline point.

The double top or bottom can have any size, but preferably it should be visible to most market participants so that its impact is bigger. Theoretically, the pattern's psychological explanation is that with the second top or bottom, the market has failed to push the prices beyond the first peak and therefore is showing weakness, which might be exploited by the seller.

There are other patterns that are more objective in nature; that is, they have clear rules of detection and initiation. These are all based on clear objective conditions and are not subject to the analyst's discretion. This facilitates their backtesting and evaluation.

Summary

Technical analysis offers a big selection of tools to analyze the markets either mathematically, graphically, or even psychologically (through patterns). The learning outcome of this chapter is to understand what technical analysis is and what the technical indicators are so that you are familiar with them when used as explanatory variables (as is the case in [Chapter 11](#)).

Introductory Python for Data Science

This is the final chapter before we dive into the realm of machine and deep learning. This chapter is optional for experienced Python developers but is important for anyone without a solid programming background. Understanding the intuition behind the algorithms is a great advantage, but that knowledge will not get you far if you fail to properly implement the algorithms. After all, these algorithms need to be coded to work and do not function manually, so you need to understand the basic syntax and how to manipulate and transform data.

As the book is not meant to be an A-Z guide to programming in Python, this chapter only focuses on some of the essentials and a few additional techniques that should help you smoothly navigate the subsequent chapters.

Downloading Python

Code is defined as a set of instructions designed to be executed by a computer. Generally, specific syntax is required so that the computer applies the set of instructions without errors. There are many coding languages, and they are divided into two broad categories:

Low-level coding languages

These are machine languages usually used to write operating systems and firmware. They are very difficult to read. These languages have a sizable level of control over hardware. Assembly language is an example of a low-level language.

High-level coding languages

These are user-friendly languages (with a high level of abstraction). They are generally used to code programs and software. Examples of high-level languages include Python and Julia.

The coding language used in this book is Python, a popular and versatile language with many advantages and wide adoption in the research and professional trading communities. As you have probably gathered from the chapter's title, you will get an introduction to Python and to the tools you need to start building your own scripts. But before that, you need to download Python.

A *Python interpreter* is software used to write and execute code written using Python syntax. I use Spyder. Some people may be more familiar with other interpreters such as Jupyter and PyCharm, but the process is the same. You can download Spyder from the [official website](#) or, even better, download it as part of a bigger package called [Anaconda](#), which facilitates installation and offers more tools. Note that Spyder is open source and free to use.

Figure 6-1 shows Python's console, where the output of the code appears.

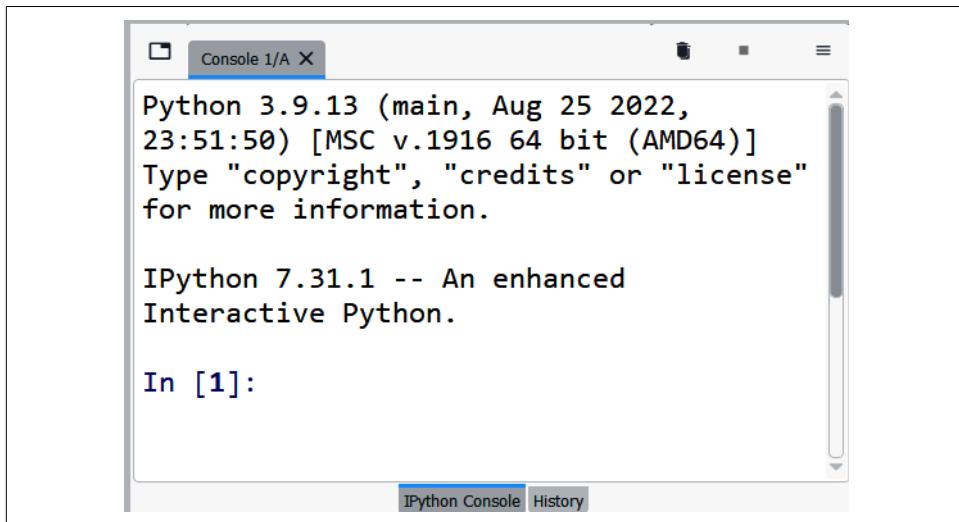


Figure 6-1. Spyder's console

Python files have the extension `.py`, and they allow you to save the code and refer to it at a later stage. You can also open multiple files of code and navigate between them.

The outline of this chapter is as follows:

- Understand the language of Python and how to write error-free code
- Understand how to use control flow and its importance with time series analysis
- Understand libraries and functions and their role in facilitating coding

- Understand different types of errors and how to handle them
- Understand how to use data manipulation libraries such as *numpy* and *pandas*
- Finally, see how to import historical financial time series data into Python so that it gets analyzed with the proper tools (those we already discussed as well as those we will discuss in coming chapters)

Basic Operations and Syntax

Syntax is the set of rules that define the structure of statements needed to write code that functions. When you are communicating with a computer, you have to make sure it understands you, so having a solid understanding of syntax is important.

Comments are nonexecutable code used to explain the executable code that follows. Comments are used so that other programmers understand the code. Comments in Python are preceded by a hash sign (#):

```
# This is a comment. Comments are ignored by the interpreter
# Comments explain the code or give more details about its use
# Comments are written on one line; otherwise, you have to rewrite '#'
```



Make sure you understand that comments are nonexecutable. This means that when you run (execute) the code, they will be ignored by the interpreter and will not return an error.

Sometimes you need to write documentation for your code, which may require multiple lines of code (even paragraphs, in some instances). Writing the hash sign at every line can be tedious and cluttersome. This is why there is a way to write long comments. To do this, write your comment between three single quotation marks as follows:

```
'''
Python was created in the late 1980s by Guido van Rossum
The name "Python" was inspired by the comedy group Monty Python
'''
```

It is worth noting that triple quotes are called *docstrings* and are not really comments (according to the official Python documentation).

Let's discuss variables and constants. A *constant* is a fixed value that does not change, whereas a *variable* takes on different values given an event. A constant can be the number 6, while a variable can be the letter *x*, which takes on any number given a set of conditions or a state. A variable is defined using the = operator:

```
# Defining a variable
x = 10
# Writing a constant
6
```

Running (executing) the previous code will store the variable `x` with its respective value in the variable explorer. Simultaneously, the output of the code will be 6. Variables are case sensitive. Therefore:

```
# Declaring my_variable
my_variable = 1
# Declaring My_variable
My_variable = 2
# The variable my_variable is different from My_variable
```

A variable declaration cannot start with a number, but a number can be included in the middle or the end of a variable declaration:

```
# Returns a SyntaxError
1x = 5
# Valid declaration
x1 = 5
# Valid declaration
x1x = 5
```

Variables can also contain underscores but nothing else:

```
# Returns a SyntaxError
x-y = 5
# Valid declaration
x_y = 5
```

It is highly recommended that variables have short and straightforward names. For example, consider creating a variable that holds the lookback period of a certain moving average (a technical indicator introduced in [Chapter 5](#)):

```
# Recommended name
ma_lookback = 10
# Not recommended name
the_lookback_on_that_moving_average = 10
```

There are several different data types with different characteristics:

Numerical data types

This is the simplest data type, formed exclusively from numbers. Numerical data types are divided into integers, float numbers, and complex numbers. *Integers* are simple whole numbers (positive or negative), such as 6 and -19. *Float numbers* are more precise than integers as they incorporate the values after the comma, for example, 2.7 and -8.09. *Complex numbers* include imaginary numbers.¹

¹ Imaginary numbers are a type of complex number that represents the square root of a negative number.

Strings

As you saw previously with comments and docstrings, it is possible to write text next to the code without it interfering with the execution process. *Strings* are text structures that represent sequences of characters. Strings can be inputs and arguments of functions and not necessarily just comments.

Booleans

A Boolean is a binary (true or false) data type used to evaluate the truth value of the given expression or condition. For example, you can use Booleans to evaluate whether the market price is above or below the 100-period moving average.

Data collection

These are sequences that contain multiple datasets, each having a different and unique usage. An *array* is a sequence of elements of the same type (mostly numerical). Arrays will be used frequently in this book (with a Python library called *numpy* that is discussed in this chapter). A *dataframe* is a two-dimensional table of structured data that is also frequently used in this book (with a Python library called *pandas* also discussed in this chapter). A *set* is a sequence of unordered elements. A *list* is an ordered collection of elements that can be of different data types. A *tuple* is an ordered, immutable collection of elements that may be of different data types. It is used for storing a fixed sequence of values. A *dictionary* represents a collection of key-value pairs grouped together.

The following code snippet shows a few examples of the numerical data type:

```
# Creating a variable that holds an integer
my_integer = 1
# Creating a variable that holds a float number
my_float_number = 1.2
# Using the built-in Python function type() to verify the variables
type(my_integer)
type(my_float_number)
```

The output should be as follows (note that the two created variables will appear in the variable explorer):

```
int # The output of type(my_integer)
float # The output of type(my_float_number)
```

Strings are simply text. The most commonly used example to explain a string is the phrase “Hello World”:

```
# Outputting the phrase Hello World
print('Hello World')
```

The output should be as follows:

```
Hello World
```

Strings can also be used as arguments in functions, as you will see later in this chapter.

Booleans are either true or false values. The following code snippet shows an example of using them:

```
# Make a statement that the type of my_integer is integer
type(my_integer) is int
# Make a statement that the type of my_float_number is float
type(my_float_number) is float
# Make a statement that the type of my_integer is float
type(my_integer) is float
"""

Intuitively, the two first statements will return True as they are indeed true. The third statement is False as the variable my_integer is an integer and not a float number
"""
```

The output of the previous code is as follows:

```
True
True
False
```

Let's discuss how operators work. You already saw an example of an operator: the assignment operator = used to define variables. Operators perform special mathematical and other tasks between variables, constants, and even data structures. There are different types of operators. Let's start with *arithmetic operators*, as shown in the following snippet:

```
# Arithmetic operator - Addition
1 + 1 # The line outputs 2
# Arithmetic operator - Subtraction
1 - 1 # The line outputs 0
# Arithmetic operator - Multiplication
2 * 2 # The line outputs 4
# Arithmetic operator - Division
4 / 2 # The line outputs 2.0 as a float number
# Arithmetic operator - Exponents
2 ** 4 # The line outputs 16
```

The next type of operator is the *comparison operators*. These are used to compare different elements. They are mostly used in control flow events, as explained in the next section of this chapter. The following snippet shows a few comparison operators:

```
# Comparison operator - Equality
2 == 2 # The line outputs True
# Comparison operator - Non equality
2 != 3 # The line outputs True
# Comparison operator - Greater than
2 > 3 # The line outputs False
# Comparison operator - Greater than or equal to
2 >= 2 # The line outputs True
# Comparison operator - Less than
2 < 3 # The line outputs True
```

```
# Comparison operator - Less than or equal to  
2 <= 2 # The line outputs True
```

Logical operators combine two or more conditions that are later evaluated. There are three logical operators: `and`, `or`, and `not`. The following code block shows an example of logical operators:

```
# Logical operator - and  
2 and 1 < 4 # The line outputs True  
2 and 5 < 4 # The line outputs False  
# Logical operator - or  
2 or 5 < 4 # The line outputs 2, which is the integer less than 4
```

Data collection structures (arrays and dataframes) are discussed in a later section, as they require an in-depth presentation due to their complexity and unique tools. Let's end this section with code that combines what has been discussed so far:

```
# Declaring two variables x and y and assigning them values  
x = 10  
y = 2.5  
# Checking the types of the variables  
type(x) # Returns int  
type(y) # Returns float  
# Taking x to the power of y and storing it in a variable z  
z = x ** y # Returns 316.22  
# Checking if the result is greater than or equal to 100  
z >= 100 # Returns True as 316.22 >= 100
```

Control Flow

Conditional statements form the first part of what is known as *control flow* (the second part is *loops*, discussed shortly). *Conditional statements* are the ancestors of today's artificial intelligence as they only execute code if certain conditions are met.

Conditional statements are managed using `if`, `elif`, and `else`. Take the following code snippet as an example:

```
# Declaring the variables  
a = 9  
b = 2  
# First condition (specific)  
if a > b:  
    print('a is greater than b')  
# Second condition (specific)  
elif a < b:  
    print('a is less than b')  
# Third condition (general)  
else:  
    print('a is equal to b')
```

Therefore, conditional statements start with `if`. Then, for every new unique and specific condition, `elif` is used (which is a fusion of `else if`), until it makes sense to use the rest of the probability universe as a condition on its own, which is used by the `else` statement. Note that the `else` statement does not need a condition, as it exists to cover the rest of the uncovered universe.

Loops are used to execute blocks of code repeatedly until a predefined condition is met. Loops are heavily used with time series to calculate indicators, verify states, and backtest trading strategies.

Loops are managed using `for` (for iterating over a finite and defined sequence or a range of elements) and `while` (used to continue the iteration until a condition is met) statements. For example, the following code prints the values {1, 2, 3, 4} using a loop:

```
# Using a for loop
for i in range(1, 5):
    print(i)
# Using a while loop
i = 1
while i < 5:
    print(i)
    i = i + 1
```

The `for` loop when translated is simply saying that for every element called `i` (or any other letter depending on the coder) in the range that starts at 1 and ends at 5 (excluded), print the value of `i` at every loop (hence, in the first loop the value of `i` is equal to 1, and in the second loop it is equal to 2).

The `while` loop says that, starting from a value of `i = 1`, while looping, print its value and then add 1 to it before finishing the first loop. End the loop when `i` becomes greater than 4.



Theoretically, a `while` loop is infinite until told otherwise.

It is worth noting that `i = i + 1` can also be expressed as `i += 1`. The goal of an algorithm is the ability to apply many operations recursively in an objective way, which makes loops extremely useful, especially when combined with conditional statements. Let's look at an example of a financial time series:

1. Create a range of values to simulate hypothetical prices.
2. Loop through the range of the data while creating the condition that if the price rose since the preceding period, print 1. Similarly, if the price fell since the preceding period, print -1. Last, print 0 if the price didn't change from the preceding period.

This can be done with the following code block:

```
# Creating the time series
time_series = [1, 3, 5, 2, 4, 1, 6, 4, 2, 4, 4, 4]
for i in range(len(time_series)):
    # The condition where the current price rose
    if time_series[i] > time_series[i - 1]:
        print(1)
    # The condition where the current price fell
    elif time_series[i] < time_series[i - 1]:
        print(-1)
    # The condition where the current price hasn't changed
    else:
        print(0)
```

The code defines a list of values (in this case, a time series called `time_series`), then loops around its length using the `len()` function to apply the conditions. Notice how at every loop, the current time step is referred to as `i`, thus making the previous time step `i - 1`.

Libraries and Functions

A *library* in Python is a group of prewritten code that offers functionality to facilitate the creation of applications. *Modules*, which are individual Python files with reusable code and data that can be imported and used in other Python code, are commonly found in libraries. A module is therefore a single Python file that contains functions and other types of code that may be used and imported by other Python programs. Large codebases are often easier to manage and maintain by using modules to divide similar code into different files.

Coding is all about simplifying tasks and making them clearer. Functions are essential in this regard. A *function* is a block of reusable code that performs a specific task when called. It only needs to be defined once. When you have a recurring task such as calculating a moving average of a time series, you can use a function so that you do not have to write the moving-average code all over again every time you want to use it. Instead, you define the function with the original code and then call it whenever you need to calculate the moving average.

Multiple functions form a module, and multiple modules form a library. A library is generally theme oriented. For example, in this book the *sklearn* library will be used with machine learning models. Similarly, data manipulation and importing are done using *numpy* and *pandas*, two libraries discussed in a later section of this chapter. Plotting and charting are done using the *matplotlib* library.

Libraries must be imported to the Python interpreter before you can use them (this is the equivalent of acknowledging their existence). The syntax for doing this is as follows:

```
# The import statement must be followed by the name of the library
import numpy
# Optionally, you can give the library a shortcut for easier reference
import numpy as np
```

Sometimes you need to import just one function or module from a library. For this, you don't need to import the entire library:

```
# Importing one function from a library
from math import sqrt
```

The preceding code says that *math* is a Python library that harbors many mathematical functions, namely the *sqrt* function, which is used to find the square root of a given number.

Let's see how to define a function. A function is defined using `def` followed by the name of the function and any optional arguments. The following example creates a function that sums any two given variables:

```
# Defining the function sum_operation and giving it two arguments
def sum_operation(first_variable, second_variable):
    # Outputting the sum of the two variables
    print(first_variable + second_variable)
# Calling the function with 1 and 3 as arguments
sum_operation(1, 3) # The output of this line is 4
```



Calling a function means executing what it's supposed to do. In other words, calling a function is simply using it. The timeline of a function is getting defined and then getting called.

Let's see how to import a function from a library and use its functions:

```
# Importing the library
import math
# Using the natural logarithm function
math.log(10)
# Using the exponential function (e)
math.exp(3)
# Using the factorial function
math.factorial(50)
```

As a side note, the *factorial* operation is a mathematical operation that is used to calculate the product of all positive integers from 1 up to a certain number (which is the argument requested in `math.factorial()`).

Libraries may not be as easy as one plus one. Sometimes external libraries require installation first before they can be imported to the Python interpreter. Installation can be done through the prompt using the following syntax:

```
pip install library_name
```

Recall [Chapter 3](#), where the maximal information coefficient (MIC) was discussed. To calculate the MIC, you can use the following code (after having defined the sine and cosine waves):

```
# Importing the library
from minepy import MINE
# Calculating the MIC
mine = MINE(alpha = 0.6, c = 15)
mine.compute_score(sine, cosine)
MIC = mine.mic()
print('Correlation | MIC: ', round(MIC, 3))
```

Importing the library directly will likely lead to an error as it has not been `pip` installed. Therefore, you must install the library first using the following syntax at the prompt (not in the Python interpreter):

```
pip install minepy
```



You may need to update Microsoft Visual C++ (to version 14.0 or greater) to avoid any errors in trying to run the *minepy* library.

It is also important to read the documentation that comes with libraries in order to use them correctly. A library's documentation will explain the aim of the functions as well as what types of arguments each function can accept (e.g., strings or numerics).

Let's return now to the topic of functions. A function can have a `return` statement that allows the result to be stored in a variable so that it can be used in other parts of the code.

Let's take two simple examples and then discuss them step by step:

```
# Defining a function to sum two variables and return the result
def sum_operation(first_variable, second_variable):
    # The summing operation is stored in a variable called final_sum
    final_sum = first_variable + second_variable
    # The result is returned
    return final_sum
# Create a new variable that holds the result of the function
```

```
summed_value = sum_operation(1, 2)
# Use the new variable in a new mathematical operation
double_summed_value = summed_value * 2
```

The previous code defines the `sum_operation` function with two arguments, then stores the operation in a variable called `final_sum` before returning it so that it can be stored externally. Next, a new variable called `summed_value` is defined as the output of the function. Finally, another variable is created, `double_summed_value`, that is the result of `summed_value` multiplied by 2. This is an example of how to use results from functions as variables in external operations. Now let's consider an example of a nested function (while keeping in mind the previously defined `sum_operation` function):

```
# Defining a function to square the result gotten from sum_operation()
def square_summed_value(first_variable, second_variable):
    # Calling the nested sum_operation function and storing its result
    final_sum = sum_operation(first_variable, second_variable)
    # Creating a variable that stores the square of final_sum
    squared_sum = final_sum ** 2
    # The result is returned
    return squared_sum
# Create a new variable that holds the result of the function
squared_summed_value = square_summed_value(1, 2)
```

The preceding code snippet defines a function called `square_summed_value`, which takes on two arguments. Furthermore, it uses a nested function, which in this case is `sum_operation`. The result of the nested function is once again stored in a variable called `final_sum`, which is used as an input in finding the `squared_sum` variable. The variable is found as `final_sum` to the power of two.

Let's end the section with common libraries in Python and machine learning (other than `numpy` and `pandas`):

- `matplotlib`: For plotting and visualizing data
- `sklearn`: For machine learning models
- `scipy`: For scientific computing and optimization
- `keras`: For neural networks
- `math`: For using mathematical tools such as square roots
- `random`: For generating random variables
- `requests`: For making HTTP requests used in web scraping

Exception Handling and Errors

Quite often, errors occur when the code is executed and the interpreter finds an obstacle that prevents it from continuing further. The most basic error is the **Syntax Error**, which occurs when there are misspelled words or missing elements that make the code unintelligible:

```
# Will not output a SyntaxError if executed
my_range = range(1, 10)
# Will output a SyntaxError if executed
my_range = range(1, 10
```

As you can see from the previous code, there is a missing parenthesis at the end of the second code line, which is not understood by the interpreter. This type of error is likely to be your most common one. Another common error is the **NameError**, which occurs when failing to define a variable before executing a code that contains it. Consider the following example:

```
x + y
```

The previous code will give you a **NameError** because the interpreter does not know the value of x and y since they were not defined.

The **ModuleNotFoundError** occurs when the interpreter cannot find the library or module you are trying to import. This generally occurs when the module or library is installed in a bad directory or when it is not properly installed. Common fixes for this issue include:

- Verifying that the module's name was written correctly
- Verifying that the module was correctly pip installed
- Verifying that the module was installed in the correct location

Another type of common error is the **TypeError**, and it occurs when you apply a certain operation on an incompatible element, such as summing an integer with a string. The following operation raises a **TypeError**:

```
# Defining variable x
x = 1
# Defining variable y
y = 'Hello
# Summing the two variables, which will raise a TypeError
x + y
```

In time series analysis, you will likely to encounter these four errors:

IndexError

This is raised when referring to an index that is out of range regarding the current array or dataframe. Imagine having an array of 300 values (rows). If you want to loop through them and, at each loop, input the number 1 in the next cell (time step + 1), the interpreter will raise an **IndexError** because in the last loop there is no next cell.

ValueError

This is raised when you try to call a function with an invalid argument. An example of this would be trying to pass an integer element as a string when calling a function.

KeyError

This occurs when trying to access an element in a dataframe that does not exist. For example, if you have three columns in the dataframe and you refer to one that does not exist (maybe due to a syntax issue), you are likely to run into a **KeyError**.

ZeroDivisionError

This error is intuitive and occurs when trying to divide a number by zero.

There are other types of errors that you may encounter. It is important to understand what they refer to so that you are able to fix them and get the code running again.

Exceptions are errors that may not be fatal to the code in the sense that they only show a warning but don't necessarily terminate the code. Therefore, exceptions occur during code execution (as opposed to errors, which occur because the interpreter is unable to execute the code). To ignore certain exceptions (and errors), the `try` and `except` keywords are used. This is useful when you are certain that handling the exception will not alter the output of the code.

Let's take an example of creating a function that divides the first column of a time series by the next value of the second column. The first step is to define the time series as a dataframe or as an array (or any other data collection structure):

```
# Importing the required library to create an array
import numpy as np
# Creating a two-column list with 8 rows
my_time_series = [(1, 3),
                  (1, 4),
                  (1, 4),
                  (1, 6),
                  (1, 4),
                  (0, 2),
                  (1, 1),
                  (0, 6)]
# Transforming the list into an array
my_time_series = np.array(my_time_series)
```

Now let's write the division function that will take any value in the first column and divide it by the next value in the second column:

```
# Defining the function
def division(first_column, second_column):
    # Looping through the length of the created array
    for i in range(len(my_time_series)):
        # Division operation and storing it in the variable x
        x = my_time_series[i, first_column] /
            my_time_series[i + 1, second_column]
        # Outputting the result
        print(x)
# Calling the function
division(0, 1)
```

Running the two previous code blocks will give an `IndexError` because in the last loop, the function cannot find the next value of the second column since it does not exist:

```
IndexError: index 8 is out of bounds for axis 0 with size 8
```

Fixing this through `try` and `except` will ignore the last calculation that is causing the problem and will return the expected results:

```
# Defining the function
def division(first_column, second_column):
    # Looping through the length of the created array
    for i in range(len(my_time_series)):
        # First part of the exception handling
        try:
            # Division operation and storing it in the variable x
            x = my_time_series[i, first_column] /
                my_time_series[i + 1, second_column]
            # Outputting the result
            print(x)
        # Exception handling of a specific error
        except IndexError:
            # Ignoring (passing) the error
            pass
# Calling the function
division(0, 1)
```

The output is as follows:

```
0.25
0.25
0.16
0.25
0.50
0.00
0.16
```

Data Structures in numpy and pandas

You now understand what a library is and know that *numpy* and *pandas* are the go-to libraries to manipulate, handle, and import data in Python. This section discusses the differences between the two, along with key functions that are definitely a great addition to your data analysis toolbox. But first, let's define these two libraries:

numpy

Short for numerical Python, *numpy* is a Python library that allows working with multidimensional arrays and matrices. It provides a powerful interface for performing various operations on arrays and matrices.

pandas

Short for panel data, *pandas* is a Python library that allows working with dataframes (a type of tabular data). It provides two main data structures: series and dataframes. A *series* is a one-dimensional array-like object that can hold any data type. A *dataframe* is a two-dimensional table-like structure that consists of rows and columns (similar to a spreadsheet).

Both libraries are very useful in analyzing time series data. Arrays hold only numerical type data and therefore do not really hold date type data. This may be one of the advantages of using *pandas* over *numpy*, but both have strengths and relative weaknesses. In the end, it is a matter of choice. This book will prioritize using *numpy* due to its simplicity and the fact that the machine learning models in the next chapter use the *sklearn* library, which is applied on arrays.



Switching between *numpy* and *pandas* requires converting the time series type. This is a relatively easy task, but it can sometimes cause the loss of certain types of data (e.g., date data).

Let's import both libraries before discussing some of their potential:

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

The following code creates two time series with two columns and three rows. The first time series is called `my_data_frame` and is created using the *pandas* library function `pd.DataFrame`. The second time series is called `my_array` and is created using the *numpy* library function `np.array`:

```
# Creating a dataframe  
my_data_frame = pd.DataFrame({'first_column' : [1, 2, 3],  
                             'second_column' : [4, 5, 6]})  
  
# Creating an array  
my_array = np.array([[1, 4], [2, 5], [3, 6]])
```

As shown in **Figure 6-2**, dataframes have real indices and can have column names. Arrays can only hold one type of data.

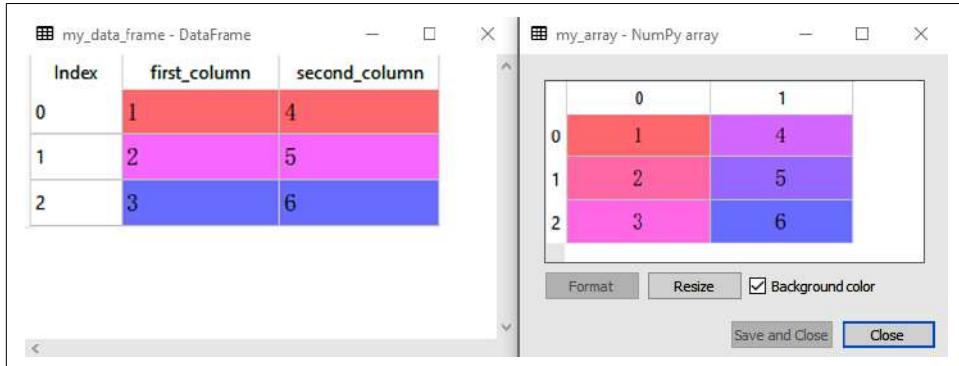


Figure 6-2. A pandas *dataframe* (left) and a numpy *array* (right)

To switch between the two types of data, you will be using the same two functions used in the previous code block:

```
# To transform my_data_frame into my_new_array  
my_new_array = np.array(my_data_frame)  
# To transform my_array into my_new_data_frame  
my_new_data_frame = pd.DataFrame(my_array)
```

Let's now take a look at useful functions that will come in handy when dealing with models. Slicing and concatenation are among the processes you must master to smoothly navigate through data analysis. Consider the following arrays:

```
first_array = np.array([ 1,  2,  3,  5,  8, 13])  
second_array = np.array([21, 34, 55, 89, 144, 233])
```

Concatenation is the act of fusing two datasets together either through rows ($\text{axis} = 0$) or through columns ($\text{axis} = 1$). Let's do both of them:

```
# Reshaping the arrays so they become dimensionally compatible  
first_array = np.reshape(first_array, (-1, 1))  
second_array = np.reshape(second_array, (-1, 1))  
# Concatenating both arrays by columns  
combined_array = np.concatenate((first_array, second_array), axis = 1)  
# Concatenating both arrays by rows  
combined_array = np.concatenate((first_array, second_array), axis = 0)
```

Now let's do the same thing for dataframes. Consider the following dataframes:

```
first_data_frame = pd.DataFrame({'first_column' : [ 1,  2,  3],  
                                'second_column' : [ 4,  5,  6]})  
second_data_frame = pd.DataFrame({'first_column' : [ 7,  8,  9],  
                                 'second_column' : [10, 11, 12]})
```

Concatenation is useful when you want to combine data into one structure. This is how it can be done with dataframes (notice that it's simply a change of syntax and function source):

```
# Concatenating both dataframes by columns
combined_data_frame = pd.concat([first_data_frame, second_data_frame],
                                axis = 1)
# Concatenating both dataframes by rows
combined_data_frame = pd.concat([first_data_frame, second_data_frame],
                                axis = 0)
```

Remember that with time series, *rows* (horizontal cells) represent one time step (e.g., hourly) with all the data inside, while *columns* represent the different types of data (e.g., open price and close price of a financial instrument). Now let's see slicing techniques for arrays:

```
# Defining a one-dimensional array
my_array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# Referring to the first value of the array
my_array[0] # Outputs 1
# Referring to the last value of the array
my_array[-1] # Outputs 10
# Referring to the sixth value of the array
my_array[6] # Outputs 7
# Referring to the first three values of the array
my_array[0:3] # Outputs array([1, 2, 3])
my_array[:3] # Outputs array([1, 2, 3])
# Referring to the last three values of the array
my_array[-3:] # Outputs array([8, 9, 10])
# Referring to all the values as of the second value
my_array[1:] # Outputs array([2, 3, 4, 5, 6, 7, 8, 9, 10])
# Defining a multidimensional array
my_array = np.array([[ 1,  2,  3,  4,  5],
                    [ 6,  7,  8,  9, 10],
                    [11, 12, 13, 14, 15]])
# Referring to the first value and second column of the array
my_array[0, 1] # Outputs 2
# Referring to the last value and last column of the array
my_array[-1, -1] # Outputs 15
# Referring to the third value and second-to-last column of the array
my_array[2, -2] # Outputs 14
# Referring to the first three values and fourth column of the array
my_array[:, 2:4] # Outputs array([[3, 4], [8, 9], [13, 14]])
# Referring to the last two values and fifth column of the array
my_array[-2:, 4] # Outputs array([10, 15])
# Referring to all the values and all the columns up until the second row
my_array[:2, :] # Outputs array([[ 1,  2,  3,  4,  5], [ 6,  7,  8,  9, 10]])
# Referring to the last row with all the columns
my_array[-1:, :] # Outputs array([[11, 12, 13, 14, 15]])
```



It is important to know that Python indexing starts at zero. This means that to refer to the first element in a data structure, you refer to its index as index = 0. It is also worth noting that in ranges, the last element is excluded, which means that the first three elements in a data structure are referred to as [0, 3], which will give the elements indexed at 0, 1, and 2.

Let's see the same thing for dataframes so that this section becomes a sort of mini reference whenever you want to manipulate data structures:

```
# Defining a one-dimensional dataframe
my_df= pd.DataFrame({'first_column': [1, 2, 3, 4, 5,
                                         6, 7, 8, 9, 10]})

# Referring to the first value of the dataframe
my_df.iloc[0]['first_column'] # Outputs 1
# Referring to the last value of the dataframe
my_df.iloc[-1]['first_column'] # Outputs 10
# Referring to the sixth value of the dataframe
my_df.iloc[6]['first_column'] # Outputs 7
# Referring to the first three values of the dataframe
my_df.iloc[0:3]['first_column'] # Outputs ([1, 2, 3])
# Referring to the last three values of the dataframe
my_df.iloc[-3:]['first_column'] # Outputs ([8, 9, 10])
# Referring to all the values as of the second value
my_df.iloc[1:]['first_column'] # Outputs ([2, 3, 4, 5, 6, 7, 8, 9, 10])

# Defining a multidimensional dataframe
my_df = pd.DataFrame({'first_column' : [ 1, 6, 11],
                      'second_column' : [ 2, 7, 12],
                      'third_column' : [ 3, 8, 13],
                      'fourth_column' : [ 4, 9, 14],
                      'fifth_column' : [ 5, 10, 15]})

# Referring to the first value and second column of the dataframe
my_df.iloc[0]['second_column'] # Outputs 2
# Referring to the last value and last column of the dataframe
my_df.iloc[-1]['fifth_column'] # Outputs 15
# Referring to the third value and second-to-last column of the dataframe
my_df.iloc[2]['fourth_column'] # Outputs 14
# Referring to the first three values and fourth column of the dataframe
my_df.iloc[:][['third_column', 'fourth_column']]
# Referring to the last two values and fifth column of the dataframe
my_df.iloc[-2:]['fifth_column'] # Outputs ([10, 15])
# Referring to all the values and all the columns up until the second row
my_df.iloc[:2] # Outputs ([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
# Referring to the last row with all the columns
my_df.iloc[-1:,:] # Outputs ([[11, 12, 13, 14, 15]])
```



Try going back to the earlier chapters to execute the code given there. You should have a more solid understanding by now.

Importing Financial Time Series in Python

This section discusses a key aspect of deploying machine and deep learning algorithms. It deals with the historical OHLC data that is needed to run the models and evaluate their performance.

The first step is to prepare the environment and everything else necessary for the success of the algorithms. For this, you need two programs:

- A Python interpreter that you use to write and execute code. You already completed this step.
- Charting and financial software that you use as a database. This part is covered in this section.

For charting benchmarks, I use MetaTrader 5, a program used by many traders around the globe. MetaTrader 5 works with Spyder, so you should start by downloading Spyder and familiarizing yourself with how it works.

From the [official website](#), download and install MetaTrader 5. You need to create a demo account, which is simply a virtual account with imaginary money. The word *demo* does not refer to a limited duration of use but to the fact that it is not using real money.

To open an account, select File > Open an Account, choose MetaQuotes Software Corp, and then click Next. Then choose the first option to open a demo account; this will let you trade virtual money. Finally, enter some basic information such as your name, email, and account type. You will not receive a verification request or any type of confirmation as the demo should launch directly, allowing you to see the charts.

[Figure 6-3](#) shows the platform's interface. By default, MetaTrader 5 does not show all the markets it covers, so you need to make them accessible for import and visualization if necessary. Click View, click Market Watch, and then right-click any of the symbols shown in the new tab and choose Show All. This way, you can see the extended list with more markets.

Before you can start coding, you need to install the MetaTrader 5 Python integration library so that you can use it later in Spyder. This is easy and requires one step. Open the Anaconda prompt and type:

```
pip install MetaTrader5
```

Installation is the bridge that allows you to use Python modules and functions designed for MetaTrader 5 in the interpreter.



Figure 6-3. MetaTrader 5 interface

The following code block uses the `import` built-in statement, which calls for internal (self-created) or external (created by third parties) libraries. You'll recall that a library is a store of functions, and thus, you need to import the libraries that are pertinent to what you want to do. For demonstration purposes, import the following modules, packages, and libraries:

```
import datetime # Gives tools for manipulating dates and time
import pytz # Offers cross-platform time zone calculations
import MetaTrader5 as mt5 # Importing the software's library
import pandas as pd
import numpy as np
```

The next step is to create the universe of the time frames that you will be able to import. Even though I will be showing you how to analyze and backtest hourly data, you can define a wider universe, as shown in the following code snippet:

```
frame_M15 = mt5.TIMEFRAME_M15          # 15-minute time frame
frameframe_M30 = mt5.TIMEFRAME_M30 # 30-minute time frame
frame_H1 = mt5.TIMEFRAME_H1           # Hourly time frame
frame_H4 = mt5.TIMEFRAME_H4           # 4-hour time frame
frame_D1 = mt5.TIMEFRAME_D1           # Daily time frame
```

```
frame_W1 = mt5.TIMEFRAME_W1           # Weekly time frame
frame_M1 = mt5.TIMEFRAME_MN1          # Monthly time frame
```



The full code is found in this book's [GitHub repository](#) under the name *master_function.py*.

A *time frame* is the frequency with which you record the prices. With hourly data, you will record the last price printed every hour. This means that in a day, you can have up to 24 hourly prices. This allows you to see the intraday evolution of the price. The aim is to record the totality of the OHLC data within a specific period.

The following code defines the current time, which is used so that the algorithm has a reference point when importing the data. Basically, you are creating a variable that stores the current time and date:

```
now = datetime.datetime.now()
```

Let's now proceed to defining the universe of the financial instruments you want to backtest. In this book, the backtests will be done exclusively on the foreign exchange (FX) market. So let's create a variable that stores some key currency pairs:

```
assets = ['EURUSD', 'USDCHF', 'GBPUUSD', 'USDCAD']
```

Now that you have your time and asset variables ready, all you need to do is create the structure of the importing algorithm. The `get_quotes()` function does this:

```
def get_quotes(time_frame, year = 2005, month = 1, day = 1,
               asset = "EURUSD"):
    if not mt5.initialize():
        print("initialize() failed, error code =", mt5.last_error())
        quit()
    timezone = pytz.timezone("Europe/Paris")
    time_from = datetime.datetime(year, month, day, tzinfo = timezone)
    time_to = datetime.datetime.now(timezone) + datetime.timedelta(days=1)
    rates = mt5.copy_rates_range(asset, time_frame, time_from, time_to)
    rates_frame = pd.DataFrame(rates)
    return rates_frame
```

Notice that in the `get_quotes()` function, you use the `pytz` and `pandas` libraries. The function starts by defining the Olson time zone,² which you can set yourself. Here is a brief, nonexhaustive list of what you can enter depending on your time zone:

² Named after its creator, Arthur David Olson, the Olson time zone addresses issues related to time zone data and daylight saving time rules. It has been an essential resource for developers and computer systems to accurately handle time-related functions and conversions.

```
America/New_York  
Europe/London  
Europe/Paris  
Asia/Tokyo  
Australia/Sydney
```

Next define two variables called `time_from` and `time_to`:

- The `time_from` variable contains the `datetime` referring to the beginning of the import date (e.g., 01-01-2020).
- The `time_to` variable contains the `datetime` referring to the end of the import date, which uses the `now` variable to represent the current time and date.

The next step is to create a variable that imports the financial data using the time periods you have specified. This is done through the `rates` variable using the `mt5.copy_rates_range()` function. Finally, using *pandas*, transform the data into a dataframe. The final function required for the importing process is the `mass_import()` function. It lets you choose the time frame using the variable and then uses the `get_quotes()` function to import the data and format it to an array. The following code snippet defines the `mass_import()` function:

```
def mass_import(asset, time_frame):  
    if time_frame == 'H1':  
        data = get_quotes(frame_H1, 2013, 1, 1, asset = assets[asset])  
        data = data.iloc[:, 1:5].values  
        data = data.round(decimals = 5)  
    return data
```

The `mass_import()` function automatically converts the dataframe into an array, so you do not have to worry about conversion when using the automatic import.



You may need to adjust the year argument higher to get the data in case you have an empty array. For instance, if you get an empty array using the `mass_import()` function, try putting a more recent year in the `get_quotes()` function ("2014" instead of "2013").

To import the historical hourly EURUSD data from the beginning of 2014 to date, you may type the following (assuming `get_quotes()`, `now`, the frames, and the libraries are already defined):

```
# Defining the universe of currency pairs  
assets = ['EURUSD', 'USDCHF', 'GBPUSD', 'USDCAD']  
# Redefining the mass_import function to switch to a default 2014  
def mass_import(asset, time_frame):  
    if time_frame == 'H1':  
        data = get_quotes(frame_H1, 2014, 1, 1, asset = assets[asset])  
        data = data.iloc[:, 1:5].values
```

```
data = data.round(decimals = 5)
# Calling the mass_import function and storing it in a variable
eurusd_data = mass_import(0, 'H1')
```



Notice how the `return` statement is used in the `mass_import` function to store the historical data in chosen variables.

Even though there is a macOS version of MetaTrader 5, the Python library only works on Windows. It requires a Windows emulator on macOS. For macOS or Linux users, you may want to try the manual import method (or the alternative way proposed in [Chapter 7](#) that uses a library called *pandas-datareader*).

Automatic import is a huge time-saver, but even Windows users may run into frustrating errors. Therefore, I will show you the manual import method, which you can use as a fix. On the [Github page](#) for this book, you will find a folder called *Historical Data*. Inside the folder there is a selection of historical financial time series in Excel format that you can download.

Manual import requires an Excel file with OHLC data that you have downloaded from a third party (such as the Excel files provided in this book's GitHub repository). In this case, you can use the *pandas* library to import it and transform it into an array.

Let's take an example of *Daily_GBPUSD_Historical_Data.xlsx*. Download the file from the repository (found in *Historical Data*) and store it on your desktop. The Spyder directory must be in the same place as the file. In layperson's terms, this means Spyder must search your desktop for the Excel file. To choose the right directory, click the folder button next to the arrow. The Directory tab should look like [Figure 6-4](#).



Figure 6-4. The Directory tab after choosing the correct folder

You should get a separate window where you can choose the desktop location and then validate the choice. Having done this, the tab should look like [Figure 6-5](#).



Figure 6-5. The Directory tab after choosing the correct desktop location

You use the `read_excel()` function (built into *pandas* and accessible after importing it) to get the values inside the Excel file. Follow this syntax:

```
# Importing the excel file into the Python interpreter  
my_data = pd.read_excel('Daily_GBPUSD_Historical_Data.xlsx.xlsx')
```

You should have a dataframe called *Daily_GBPUSD_Historical_Data.xlsx* with five different columns representing open, high, low, and close prices. You generally have to enter the library's name before using a function that belongs to it; this is why `read_excel()` is preceded by `pd`.



I recommend using automatic import for Windows users and manual import for macOS users due to compatibility issues.

Summary

Python, a major star among coding languages, enjoys widespread adoption by the developer community. Mastering it is key to unlocking huge potential in the data science world.

The next chapter discusses machine learning and different prediction algorithms. The main aim is to be able to code the algorithms and run a backtest over financial data. You will see that once you start understanding the process, it becomes a matter of removing one algorithm and plugging another one in (in case they have the same assumptions). The warm-up chapters are over, and it's time to start coding.

Machine Learning Models for Time Series Prediction

Machine learning is a subfield of AI that focuses on the development of algorithms and models that enable computers to learn and make predictions or decisions without being explicitly programmed, hence the term *learning*. Machine learning deals with the design and construction of systems that can automatically learn and improve from experience, typically by analyzing and extracting patterns from large amounts of data.

This chapter presents the framework of using machine learning models for time series prediction and discusses a selection of known algorithms.

The Framework

The *framework* is very important, as it organizes the way the whole research process is done (from data collection to performance evaluation). Having a proper framework ensures harmony across the backtests, which allows for proper comparison among different machine learning models. The framework may follow these chronological steps:

1. Import and preprocess the historical data, which must contain a sufficient number of values to ensure a decent backtest and evaluation.
2. Perform a *train-test* split, which splits the data into two parts where the first part of the data (e.g., from 2000 to 2020) is reserved for training the algorithm so that it understands the mathematical formula to predict the future values, and the second part of the data (e.g., from 2020 to 2023) is reserved for testing the algorithm's performance on data that it has never seen before.

3. Fit (train) and predict (test) the data using the algorithm.
4. Run a performance evaluation algorithm to understand the model's performance in the past.



The *training set* is also called the *in-sample data*, and the *test set* is also called the *out-of-sample data*.

The first step of the framework was discussed in [Chapter 6](#). You should now be able to easily import historical data using Python. The train-test split divides the historical data into a training (in-sample) set where the model is fitted (trained) so that an implied forecasting function is found, and a test (out-of-sample) set where the forecasting function that has been calculated on the training set is applied and evaluated. Theoretically, if the model does well on the test set, it is likely that you have a potential candidate for a trading strategy, but this is just a first step and the reality is much more complicated than that.

So that everything goes smoothly, download *master_function.py* from the [GitHub repository](#), and then set the directory of the Python interpreter (e.g., Spyder) in the same location as the downloaded file so that you can import it as a library and use its functions. For example, if you download the file to your desktop, you may want to set the directory as shown in [Figure 6-5](#) in [Chapter 6](#). The choice of the directory is typically found on the top-right corner in Spyder (above the variable explorer).



If you prefer not to import *master_function.py*, you can just open it in the interpreter as a normal file and execute it so that Python defines the functions inside. However, you have to do this every time you restart the kernel.

Now, preprocess (transform) and split the time series into four different arrays (or dataframes if you wish), with each array having a utility:

Array x_train

The in-sample set of features (i.e., independent variables) that explain the variations of the variable that you want to forecast. They are the predictors.

Array y_train

The in-sample set of dependent variables (i.e., the right answers) that you want the model to calibrate its forecasting function on.

Array x_test

The out-of-sample set of features that will be used as a test of the model to see how it performs on this never-before-seen data.

Array y_test

Contains the real values that the model must approach. In other words, these are the right answers that will be compared with the model's forecasts.

Before the split, it is important to know what is being forecasted and what is being used to forecast it. In this chapter, lagged price differences (returns) will be used for the forecast. Normally, a few tests must be made before doing this, but for simplicity, let's leave them out and suppose that the last 500 daily EURUSD returns have predictive power over the current return, which means that you can find a predictive formula that uses the last 500 observations to observe the next one:

Dependent variable (forecast)

The $t+1$ return of the EURUSD in the daily time frame. This is also referred to as the y variable.

Independent variables (inputs)

The last 500 daily returns of the EURUSD. These are also referred to as the x variables.

Figure 7-1 shows the EURUSD daily returns over a certain time period. Notice its stationary appearance. According to the ADF test (seen in [Chapter 3](#)), the returns dataset seems stationary and is valid for a regression analysis.

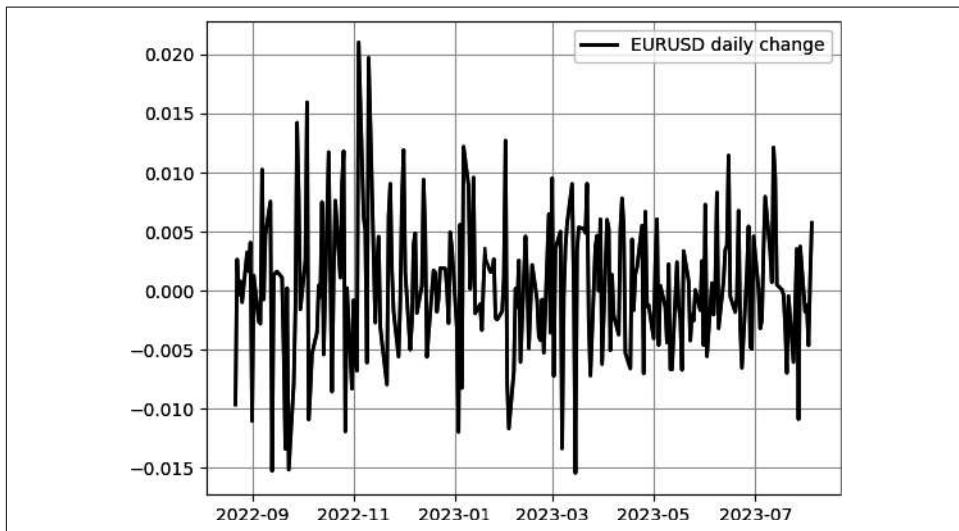


Figure 7-1. The EURUSD daily returns.



In this chapter, the features (x values) will be the lagged daily price differences of the EURUSD.¹ In subsequent chapters, the features used will be either lagged returns or values of technical indicators. Note that you can use whichever features you think are worthy of being considered as predictive.

The choice of the time frame (daily) is ideal for traders who want an intraday view that will help them trade the market and close the position before the end of the day.

Let's use the dummy regression model as a first basic example. *Dummy regression* is a comparison machine learning algorithm that is only used as a benchmark, as it uses very simple rules for predictions that are unlikely to add any real forecasting value. The real utility of the dummy regression is to see whether your real model outperforms it or not. As a reminder, the process followed by the machine learning algorithms is composed of the following steps:

1. Import the data.
2. Preprocess and split the data.
3. Train the algorithm.
4. Predict on test data using the training parameters. Also, predict on training data for comparison.
5. Plot and evaluate the results.

Start by importing the libraries required for this chapter:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from master_function import data_preprocessing, mass_import
from master_function import plot_train_test_values,
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Now import the specific library for the algorithm you will use:

```
from sklearn.dummy import DummyRegressor
```

The next step is to import and transform the close price data. Remember, you are trying to forecast daily returns, which means that you must select only the close column and then apply a differencing function on it so that prices become differenced:

¹ Price differences will be referred to as returns for simplicity. In general, returns can also represent a percentage return of a time series.

```
# Importing the differenced close price of EURUSD daily time frame
data = np.diff(mass_import(0, 'H1')[:, 3])
```



In finance, the term *returns* typically refers to the gain or loss generated by an investment or a certain asset, and it can be calculated by taking the difference between the current value of an asset and its value at a previous point in time. This is essentially a form of differencing, as you are calculating the change or difference in the asset's value.

In time series analysis, differencing is a common technique used to make time series data stationary, which can be helpful for various analyses. Differencing involves subtracting consecutive observations from each other to remove trends or seasonality, thereby focusing on the changes in the data.

Next, set the hyperparameters of the algorithm. In the case of these basic algorithms, it would be the number of lags (number of predictors) and the percentage split of data:

```
# Setting the hyperparameters
num_lags = 500
train_test_split = 0.80
```

A `train_test_split` of 0.80 means that 80% of the data will be used for training while the remaining 20% will be used for testing.

The function to split and define the four necessary arrays for the backtest can be defined as follows:

```
def data_preprocessing(data, num_lags, train_test_split):
    # Prepare the data for training
    x = []
    y = []
    for i in range(len(data) - num_lags):
        x.append(data[i:i + num_lags])
        y.append(data[i+ num_lags])
    # Convert the data to numpy arrays
    x = np.array(x)
    y = np.array(y)
    # Split the data into training and testing sets
    split_index = int(train_test_split * len(x))
    x_train = x[:split_index]
    y_train = y[:split_index]
    x_test = x[split_index:]
    y_test = y[split_index:]
    return x_train, y_train, x_test, y_test
```

Call the function to create the four arrays:

```
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data,
                                                       num_lags,
                                                       train_test_split)
```

You should now see four new arrays appearing in the variable explorer. The next step is to train the data using the chosen algorithm:

```
# Fitting the model
model = DummyRegressor(strategy = 'mean')
model.fit(x_train, y_train)
```

Note that the dummy regression can take any of the following strategies as arguments:

mean

Always predicts the mean of the training set

median

Always predicts the median of the training set

quantile

Always predicts a specified quantile of the training set, provided with the quantile parameter

constant

Always predicts a constant value that is provided by the user

As you can see from the previous code, the selected parameter is `mean`. Naturally, this signifies that all the predictions made will simply be the mean of the training set (`y_train`). This is why dummy regression is only used as a benchmark and not as a serious machine learning model.

The next step is to predict on the test data, as well as on the training data as a means of comparison. Note that the predictions on the training data have no value since the algorithm has already seen the data during training, but it is interesting to know how worse or better the algorithm performs on data that has never been seen before:

```
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

To make sure your reasoning is correct with regard to using the dummy regression algorithm, manually calculate the mean of `y_train` and compare it to the value you get in every `y_predicted`. You will see that it's the same:

```
# Comparing the mean of y_train to an arbitrary value in y_predicted
y_train.mean() == y_predicted[123]
```

The output should be as follows:

True

Finally, use the following function to plot the last training data followed by the first test data and the equivalent predicted data:

```
# Plotting
plot_train_test_values(100, 50, y_train, y_test, y_predicted)
```



You can find the definition of the `plot_train_test_values()` function in this book's [GitHub repository](#).

Figure 7-2 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`. Naturally, the dummy regression algorithm predicts a constant value, which is why the prediction line alongside the test values is a straight line.

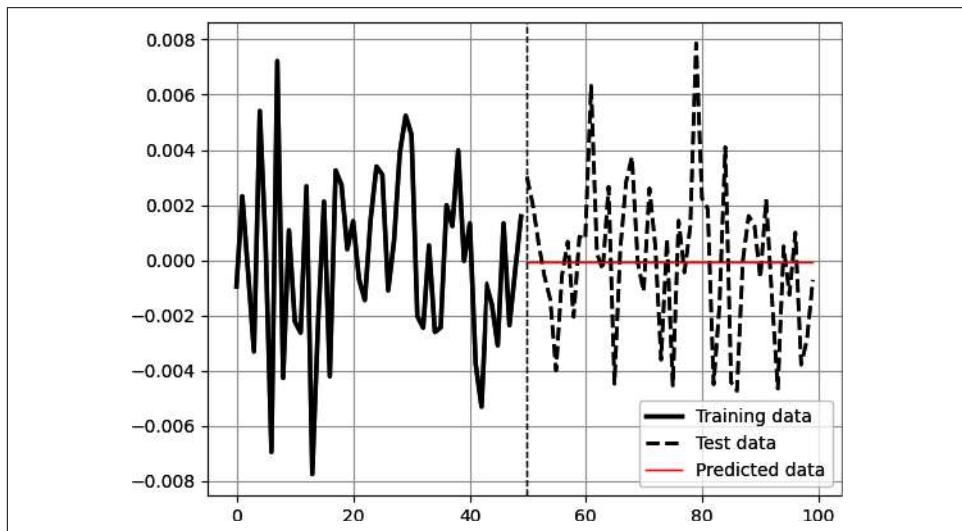


Figure 7-2. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the dummy regression algorithm.



If you want the figures to be plotted in a separate window, type `%matplotlib qt` in the console. If you want the figures to be inside the plots explorer, type `%matplotlib inline` in the console.

How can you tell whether a model is performing well or not? *Performance evaluation* is a key concept in trading and algorithmic development as it ensures that you pick the right model and take it live. However, the task is not simple, due to an ironically simple question: *If the past performance was good, what guarantees it continues to be good?*

This question is painful, but it points toward the right direction. The answer to this question is subjective. For now, let's talk about the different ways to measure the performance of a model. To simplify the task, I will split the performance and evaluation metrics into two: model evaluation and trading evaluation. *Model evaluation* deals with the algorithm's performance in its forecasts, while *trading evaluation* deals with the financial performance of a system that trades using the algorithm (an example of a trading evaluation metric is the net profit).

Let's start with model evaluation. *Accuracy* is the first metric that comes to mind when comparing forecasts to real values, especially in the financial markets. Theoretically, if you predict the direction (up or down) and you get it right, you should make money (excluding transaction costs). Accuracy is also referred to as the *hit ratio* in financial jargon and is calculated as follows:

$$\text{Accuracy} = \frac{\text{Correct predictions}}{\text{Total predictions}} \times 100$$

For example, if you made 100 predictions last year and 73 of them were correct, you would have a 73% accuracy.

Forecasting can also be evaluated by how close the predicted values ($y_{\text{predicted}}$) are to the real values (y_{test}). This is done by loss functions. A *loss function* is a mathematical calculation that measures the difference between the predictions and the real (test) values. The most basic loss function is the *mean absolute error* (MAE). It measures the average of the absolute differences between the predicted and actual values. The mathematical representation of MAE is as follows:

$$MAE = \frac{\sum_{i=1}^n |\hat{y}_i - y_i|}{n}$$

\hat{y} is the predicted value

y is the real value

Therefore, MAE calculates the average distance (or positive difference) between the predicted and real values. The lower the MAE, the more accurate the model.

The *mean squared error* (MSE) is one of the commonly used loss functions for regression. It measures the average of the squared differences between the predicted and actual values. You can think of MSE as the equivalent of the variance metric seen in [Chapter 3](#). The mathematical representation of MSE is as follows:

$$MSE = \frac{\sum_{i=1}^n (\hat{y} - y_i)^2}{n}$$

Hence, MSE calculates the average squared distance between the predicted and the real values. Similar to the MAE, the lower the MSE, the more accurate the model. With this in mind, it helps to compare apples to apples (such as with variance and standard deviation, as seen in [Chapter 3](#)). Therefore, the *root mean squared error* (RMSE) has been developed to tackle this problem (hence, scaling the error metric back to the same units as the target variable). The mathematical representation of RMSE is as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (\hat{y} - y_i)^2}{n}}$$

The RMSE is the equivalent of the standard deviation in descriptive statistics.



MAE is relatively less sensitive to outliers than MSE, and it is often used when the data contains extreme values or when the absolute magnitude of the error is more important than its squared value. On the other hand, as MSE gives more weight to larger errors, it is the go-to loss function when trying to improve the performance of the model.

When evaluating models using MAE, MSE, or RMSE, it is important to have a baseline for comparison:

- If you have built multiple regression models, you can compare their metrics to determine which model performs better. The model with the lower metric is generally considered to be more accurate in its predictions.
- Depending on the specific problem, you may have a threshold value for what is considered an acceptable level of prediction error. For example, in some cases, an RMSE below a certain threshold may be considered satisfactory, while values above that threshold may be considered unacceptable.
- You can compare the loss functions of the training data with the loss functions of the test data.

Algorithms may sometimes be directionally biased for many reasons (either structurally or externally). A *biased model* takes on significantly more trades in one direction than the other (an example would be an algorithm having 200 long positions and 30 short positions). *Model bias* measures this as a ratio by dividing the number of long positions by the number of short positions. The ideal model bias is around 1.00, which implies a balanced trading system. The mathematical representation of model bias is as follows:

$$\text{Model bias} = \frac{\text{Number of bullish signals}}{\text{Number of bearish signals}}$$

If a model has had 934 long positions and 899 short positions this year, then the model bias metric is 1.038, which is acceptable. This means that the model is not really biased. It is worth noting that a model bias of 0.0 represents the absence of any bullish signals, and a model bias that has an undefined value represents the absence of any bearish signals (due to the division by zero).

Now we'll turn our attention to trading evaluation. Finance pioneers have been developing metrics that measure the performance of strategies and portfolios. Let's discuss the most common and most useful ones. The most basic metric is the *net return*, which is essentially the return over the invested capital after a trading period that has at least one closed trade. The mathematical representation of the net return is as follows:

$$\text{Net return} = \left(\frac{\text{Final value}}{\text{Initial value}} - 1 \right) \times 100$$

The word *net* implies a result after deducting fees; otherwise, it is referred to as a *gross* return. For example, if you start the year with \$52,000 and finish at \$67,150, you would have made 29.13% (a net profit of \$15,150).

Another profitability metric is the *profit factor*, which is the ratio of the total gross profits to the total gross losses. Intuitively, a profit factor above 1.00 implies a profitable strategy, and a profit factor below 1.00 implies a losing strategy. The mathematical representation of the profit factor is as follows:

$$\text{Profit factor} = \frac{\text{Gross profits}}{\text{Gross losses}}$$

The profit factor is a useful metric for evaluating the profitability of a trading strategy because it takes into account both the profits and losses generated by the strategy, rather than just looking at one side of the equation. The profit factor of a trading strategy that has generated \$54,012 in profits and \$29,988 in losses is 1.80.

The next interesting metric relates to individual trades. The *average gain* per trade calculates the average profits (or positive returns) per trade based on historical data, and the *average loss* per trade calculates losses (or negative returns) per trade based on historical data. These two metrics give an expected return depending on the outcome. Both metrics are calculated following these formulas:

$$\text{Average gain} = \frac{\text{Total profit}}{\text{Number of winning trades}}$$

$$\text{Average loss} = \frac{\text{Total losses}}{\text{Number of losing trades}}$$

The next metric relates to risk and is one of the most important measures of evaluation. *Maximum drawdown* is a metric that measures the largest percentage decline in the value of an investment or portfolio from its highest historical peak to its lowest point. It is commonly used to assess the downside risk of an investment or portfolio. For example, if an investment has a peak value of \$100,000 and its value subsequently drops to \$50,000 before recovering, the maximum drawdown would be 50%, which is the percentage decline from the peak value to the trough. Maximum drawdown is calculated as follows:

$$\text{Maximum drawdown} = \left(\frac{\text{Trough value} - \text{Peak value}}{\text{Peak value}} \right) \times 100$$

Finally, let's discuss a well-known profitability ratio called the *Sharpe ratio*. It measures how much return is generated by units of excess risk. The formula of the ratio is as follows:

$$\text{Sharpe} = \frac{\mu - r}{\sigma}$$

μ is the net return

r is the risk-free rate

σ is the volatility of returns

So, if the net return is 5% and the risk-free rate is 2% while the volatility of returns is 2.5%, the Sharpe ratio is 1.20. Anything above 1.00 is desirable as it implies that the strategy is generating positive excess risk-adjusted return.

Risk-Free Rate Explained

The *risk-free rate* refers to the theoretical rate of return on an investment that carries no risk. It serves as a benchmark for evaluating the potential returns of other investments that do involve risk. In practice, the risk-free rate is typically based on the yield of a government-issued bond, usually one with a short-term maturity.

The specific risk-free rate can vary depending on the country and currency involved. In the United States, for example, the risk-free rate is often associated with the yield on US Treasury securities. The most commonly used benchmark is the yield on the 10-year Treasury note, as it is considered a relatively low-risk investment.

The focus of this book is on developing machine and deep learning algorithms, so the performance evaluation step will solely focus on accuracy, RMSE, and model bias (with the correlation between the predicted variables as an extra metric). The performance functions can be found in the GitHub repository, along with the complete scripts.

The model's results on the EURUSD after applying the performance metrics are as follows:

```
Accuracy Train = 49.28 %
Accuracy Test = 49.33 %
RMSE Train = 0.0076467838
RMSE Test = 0.0053250347
Model Bias = 0.0
```

With a bias of 0.0, it's easy to see that this is a dummy regression model. The bias means that according to the formula, all the forecasts are bearish. Taking a look at the details of the predictions, you will see that they are all constant values.

More Ways to Import Data

Some readers may not have the right operating system to run the importing algorithm through MetaTrader5. Fortunately, there are alternative ways to import the data. The first method is to use the *pandas_datareader* library, which does not require third-party software installation. It does, however, require a pip installation. To import the EURUSD daily, use the following code (also found in the GitHub repository):

```
# Importing the required libraries
import pandas_datareader as pdr
import numpy as np
# Set the start and end dates for the data
start_date = '2000-01-01'
end_date   = '2023-06-01'
# Fetch EURUSD price data
```

```

data = np.array((pdr.get_data_fred('DEXUSEU',
                                   start = start_date,
                                   end = end_date)).dropna())
# Difference the data and make it stationary
data = np.diff(data[:, 0])

```

The second method is to simply refer to the *Historical Data* folder in the GitHub repository, which contains multiple Excel files with the historical data. After downloading them, you can import them into Spyder using the following code:

```

# Importing the required libraries
import pandas as pd
import numpy as np
# Import the data (write the code in one line)
data = np.array(pd.read_excel('Daily_EURUSD_Historical_Data.xlsx')
               ['<CLOSE>'])
# Difference the data and make it stationary
data = np.diff(data)

```

Make sure to check that the directory of the interpreter is the same as the downloaded files; otherwise, you will get an error.



The key takeaways from this section are as follows:

- Automatic data import and creation saves you time and allows you to concentrate on the main issues of the algorithm.
- For a proper backtest, the data must be split into a training set and a test set.
- The training set contains `x_train` and `y_train`, with the former containing the values that are supposed to have a predictive power over the latter.
- The test set contains `x_test` and `y_test`, with the former containing the values that are supposed to have a predictive power (even though the model hasn't encountered them in its training) over the latter.
- Fitting the data is when the algorithm runs on the training set; predicting the data is when the algorithm runs on the test set.
- The predictions are stored in a variable called `y_predicted` that is compared to `y_test` for performance evaluation purposes.
- The main aim of the algorithms is to have good accuracy and stable, low-volatility returns.

Machine Learning Models

This section presents a selection of machine learning models using the framework developed so far. It is important to understand every model's strengths and weaknesses so that you know which model to choose depending on the forecasting task.

Linear Regression

The *linear regression* algorithm works by finding the best-fitting line that minimizes the sum of squared differences between the predicted and actual target values. The most used optimization technique in this algorithm is the *ordinary least squares* (OLS) method.²

The model is trained on the training set using the OLS method, which estimates the coefficients that minimize the sum of squared differences between the predicted and actual target values to find the optimal coefficients for the independent variables (the coefficients represent the y -intercept and the slope of the best-fitting line, respectively). The output is a linear function that gives the expected return given the explanatory variables weighted by the coefficient with an adjustment for noise and the intercept.

To import the linear regression library from *sklearn*, use the following code:

```
from sklearn.linear_model import LinearRegression
```

Now let's look at the algorithm's implementation:

```
# Fitting the model
model = LinearRegression()
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

The model assumes that the linear relationship that has held in the past will still hold in the future. This is unrealistic, and it ignores the fact that market dynamics and drivers are constantly shifting whether in the short term or the long term. They are also nonlinear.

Figure 7-3 shows the evolution of the forecasting task from the last values of *y_train* to the first values of *y_test* and *y_predicted*.

² The ordinary least squares method uses a mathematical formula to estimate the coefficients. It involves matrix algebra and calculus to solve for the coefficients that minimize the sum of squared residuals.

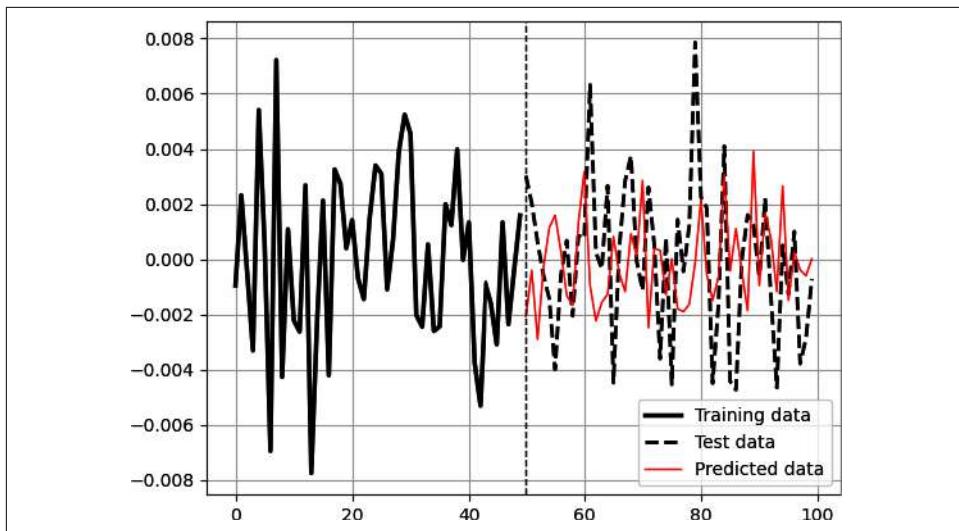


Figure 7-3. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the linear regression algorithm.

The model's results on the EURUSD after applying the performance metrics are as follows:

```

Accuracy Train = 58.52 %
Accuracy Test = 49.54 %
RMSE Train = 0.007096094
RMSE Test = 0.0055932632
Correlation In-Sample Predicted/Train = 0.373
Correlation Out-of-Sample Predicted/Test = 0.014
Model Bias = 0.93

```

The results indicate poor performance coming from the linear regression algorithm, with an accuracy below 50.00%. As you can see, the accuracy generally drops after switching to the test set. The correlation between the in-sample predictions and the real in-sample values also drops from 0.373 to 0.014. The model bias is close to equilibrium, which means that the number of long signals is close to the number of short signals.

There are a few things to note regarding the model's results:

- The transaction costs have not been incorporated, and therefore, these are gross results (not net results).
- There is no risk management system, as this is a pure time series machine learning model and not a full trading algorithm that incorporates stops and targets.

Therefore, as this is a purely directional model, the job is to try to maximize the number of correct forecasts. With a daily horizon, you are searching for accuracy.

- Different FX data providers may have small differences in the historical data that may cause some differences between backtests.

Models are made to be optimized and tweaked. The process of optimization may include any of the following techniques:

Choosing the right predictors is paramount to a model's success

In this chapter, the predictors used are the lagged returns. This has been chosen arbitrarily and is not necessarily the right choice. Predictors must be chosen based on economic and statistical intuition. For example, it may be reasonable to choose the returns of gold to explain (predict) the variations on the S&P 500 index as they are economically linked. Safe haven assets like gold rise during periods of economic uncertainty, while the stock market tends to fall. This negative correlation may harbor hidden patterns between the returns of both instruments. Another way of choosing predictors is to use technical indicators such as the relative strength index (RSI) and moving averages.

Proper splitting is crucial to evaluate the model properly

Train-test splits are important as they determine the window of evaluation. Typically, 20/80 and 30/70 are used, which means that 20% (30%) of the data is used for the testing sample and 80% (70%) is used for the training sample.

Regularization techniques can help prevent biases

Ridge regression and Lasso regression are two common regularization methods used in linear regression. *Ridge regression* adds a penalty term to the OLS function to reduce the impact of large coefficients, while *Lasso regression* can drive some coefficients to zero, effectively performing feature selection.

The model seen in this section is called an *autoregressive model* since the dependent variable depends on its past values and not on exogenous data. Also, since at every time step, 500 different variables (with their coefficients) have been used to predict the next variable, the model is referred to as a *multiple linear regression* model. In contrast, when the model only uses one dependent variable to predict the dependent variable, it is referred to as a *simple linear regression* model.

The advantages of linear regression are:

- It is easy to implement and train. It also does not consume a lot of memory.
- It outperforms when the data has a linear dependency.

The disadvantages of linear regression are:

- It is sensitive to outliers.
- It is easily biased (more on this type of bias in “[Overfitting and Underfitting](#)” on page 211).
- It has unrealistic assumptions, such as the independence of data.

Before moving on to the next section, it is important to note that some linear regression models do not transform the data. You may see extremely high accuracy and a prediction that is very close to the real data, but the reality is that the prediction lags by one time step. This means that at every time step, the prediction is simply the last real value. Let’s prove this using the previous example. Use the same code as before, but omit the price differencing code. You should see [Figure 7-4](#).

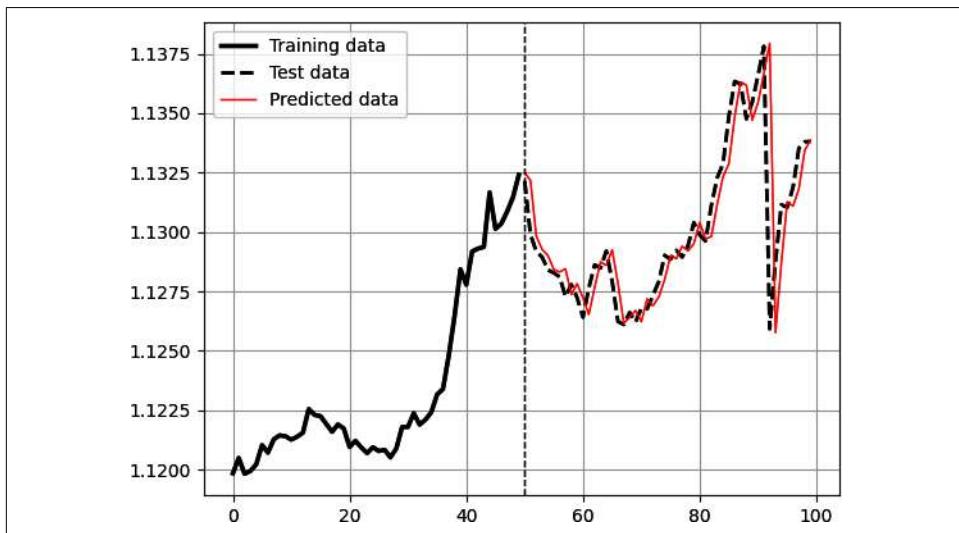


Figure 7-4. Nonstationary training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the linear regression algorithm.

Notice how it’s simply lagging the real values and not adding any predictive information. Always transform nonstationary data when dealing with such models. Nonstationary data cannot be forecasted using this type of algorithm (there are exceptions, though, which you will see later on).

Using linear regression on nonstationary data, such as market prices, and observing that the forecasts are the same as the last value might indicate an issue known as *naive forecasting*. This occurs when the most recent observation (in this case, the last value) is simply used as the forecast for the next time period. While this approach can

sometimes work for certain types of data, it is generally not a sophisticated forecasting method and may not capture the underlying patterns or trends in the data. There are a few reasons why this might happen:

Lack of predictive power

Linear regression assumes that there is a linear relationship between the independent variable(s) and the dependent variable. If the data is highly nonstationary and lacks a clear linear relationship, then the linear regression model may not be able to capture meaningful patterns and will default to a simplistic forecast like naive forecasting.

Lagging indicators

Market prices often exhibit strong autocorrelation, meaning that the current price is highly correlated with the previous price. In such cases, if the model only takes into account lagged values as predictors, it might simply replicate the last value as the forecast.

Lack of feature engineering

Linear regression models rely on the features (predictors) you provide to make forecasts. If you're using only lagged values as predictors and not incorporating other relevant features, the model might struggle to generate meaningful forecasts.

Model complexity

Linear regression is a relatively simple modeling technique. If the underlying relationship in the data is more complex than can be captured by a linear equation, the model might not be able to make accurate forecasts.

Support Vector Regression

Support vector regression (SVR) is a machine learning algorithm that belongs to the family of *support vector machines* (SVMs). SVR is specifically designed for regression problems, where the goal is to predict continuous numerical values (such as return values).

SVR performs regression by finding a hyperplane in a high-dimensional feature space that best approximates the relationship between the input features and the target variable. Unlike traditional regression techniques that aim to minimize the errors between the predicted and actual values, SVR focuses on finding a hyperplane that captures the majority of the data within a specified margin, known as the *epsilon tube* (loss function).

The key idea behind SVR is to transform the original input space into a higher-dimensional space using a kernel function. This transformation allows SVR to implicitly map the data into a higher-dimensional feature space, where it becomes

easier to find a linear relationship between the features and the target variable. The kernel function calculates the similarity between two data points, enabling the SVR algorithm to work effectively in nonlinear regression problems. The steps performed in the SVR process are as follows:

1. The algorithm employs a kernel function to transform the input features into a higher-dimensional space. Common kernel functions include the linear kernel, polynomial kernel, radial basis function (RBF) kernel, and sigmoid kernel. The choice of kernel depends on the data and the underlying problem.
2. The algorithm then aims to find the hyperplane that best fits the data points within the epsilon tube. The training process involves solving an optimization problem to minimize the error (using a loss function such as MSE) while controlling the margin.



The RBF kernel is a popular choice for SVR because it can capture nonlinear relationships effectively. It is suitable when there is no prior knowledge about the specific form of the relationship. The RBF kernel calculates the similarity between feature vectors based on their distance in the input space. It uses a parameter called *gamma*, which determines the influence of each training example on the model. Higher gamma values make the model focus more on individual data points, potentially leading to errors.

By finding an optimal hyperplane within the epsilon tube, SVR can effectively capture the underlying patterns and relationships in the data, even in the presence of noise or outliers. It is a powerful technique for regression tasks, especially when dealing with nonlinear relationships between features and target variables.

As SVR is sensitive to the scale of the features, it's important to bring all the features to a similar scale. Common scaling methods include *standardization* (mean subtraction and division by standard deviation) and *normalization* (scaling features to a range, e.g., [0, 1]).

Let's take a look at SVR in action. Once again, the aim is to predict the next EURUSD return given the previous returns. To import the SVR library and the scaling library, use the following code:

```
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
```

For the SVR algorithm, a little tweaking was done to get acceptable forecasts. The tweak was to reduce the number of lagged values from 500 to 50:

```
num_lags = 50
```

This allows the SVR algorithm to improve its forecasts. You will see throughout the book that part of performing these types of backtests is tweaking and calibrating the models.

Next, to implement the algorithm, use the following code:

```
# Fitting the model
model = make_pipeline(StandardScaler(),
                      SVR(kernel = 'rbf', C = 1, gamma = 0.04,
                           epsilon = 0.01))
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

Figure 7-5 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

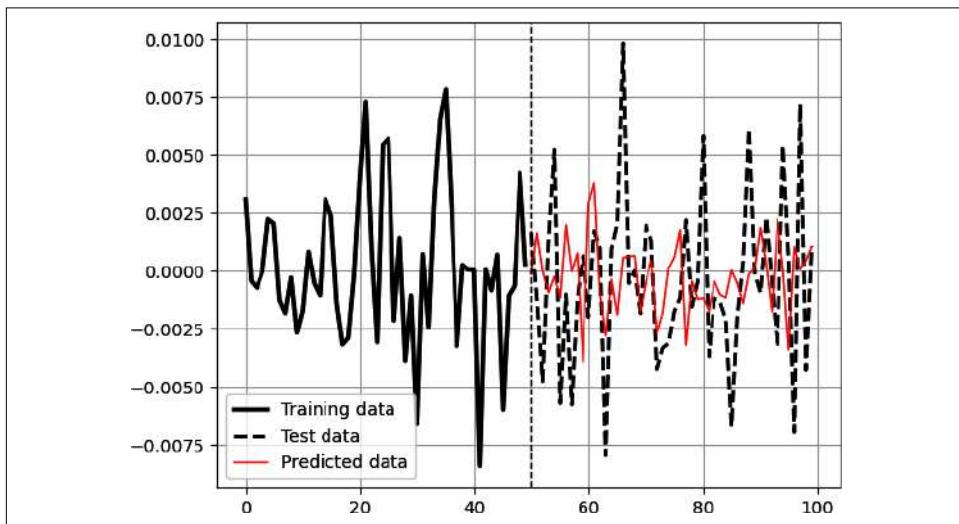


Figure 7-5. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the SVR algorithm.

The model's results are as follows:

```
Accuracy Train = 57.94 %
Accuracy Test = 50.14 %
RMSE Train = 0.0060447699
RMSE Test = 0.0054036167
Correlation In-Sample Predicted/Train = 0.686
Correlation Out-of-Sample Predicted/Test = 0.024
Model Bias = 0.98
```

The advantages of SVR are:

- It performs well even in high-dimensional feature spaces, where the number of features is large compared to the number of samples. It is particularly useful when dealing with complex datasets.
- It can capture nonlinear relationships between input features and the target variable by using kernel functions.
- It is robust to outliers in the training data due to the epsilon-tube formulation. The model focuses on fitting the majority of the data within the specified margin, reducing the influence of outliers.

The disadvantages of SVR are:

- It has several hyperparameters that need to be tuned for optimal performance. Selecting appropriate hyperparameters can be a challenging task and may require extensive experimentation.
- It can be computationally expensive, especially for large datasets or when using complex kernel functions.
- It can be sensitive to the choice of hyperparameters. Poorly chosen hyperparameters can lead to fitting issues.

Stochastic Gradient Descent Regression

Gradient descent (GD) is a general optimization algorithm used to minimize the cost or loss function of a model, and it serves as the foundation for various optimization algorithms.



Gradient simply refers to a surface's slope or tilt. To get to the lowest point on the surface, one must literally descend a slope.

Stochastic gradient descent (SGD) is an iterative optimization algorithm commonly used for training machine learning models, including regression models. It is particularly useful for large datasets and online learning scenarios. When applied to time series prediction, SGD can be used to train regression models that capture temporal patterns and make predictions based on historical data. SGD is therefore a type of linear regression that uses stochastic gradient descent optimization to find the best-fitting line.

Unlike ordinary least squares, SGD updates the model's parameters iteratively, making it more suitable for large datasets (which are treated in small batches). Instead of using the entire dataset for each update step, SGD randomly selects a small batch of samples or individual samples from the training dataset. This random selection helps to introduce randomness and avoid getting stuck in local optima (you can refer to [Chapter 4](#) for more information on optimization). The main difference between GD and SGD lies in how they update the model's parameters during optimization.



SGD does not belong to any particular family of machine learning models; it is essentially an optimization technique.

GD computes the gradients over the entire training dataset, updating the model's parameters once per epoch, while SGD computes the gradients based on a single training example or mini batch, updating the parameters more frequently. SGD is faster but exhibits more erratic behavior, while GD is slower but has a smoother convergence trajectory. SGD is also more robust to local minima. The choice between GD and SGD depends on the specific requirements of the problem, the dataset size, and the trade-off between computational efficiency and convergence behavior.

As usual, the first step is to import the necessary libraries:

```
from sklearn.linear_model import SGDRegressor  
from sklearn.preprocessing import StandardScaler  
from sklearn.pipeline import make_pipeline
```

Next, to implement the algorithm, use the following code:

```
# Fitting the model  
model = make_pipeline(StandardScaler(), SGDRegressor(max_iter = 50,  
                                                    tol = 1e-3))  
model.fit(x_train, y_train)  
# Predicting in-sample  
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))  
# Predicting out-of-sample  
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

[Figure 7-6](#) shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

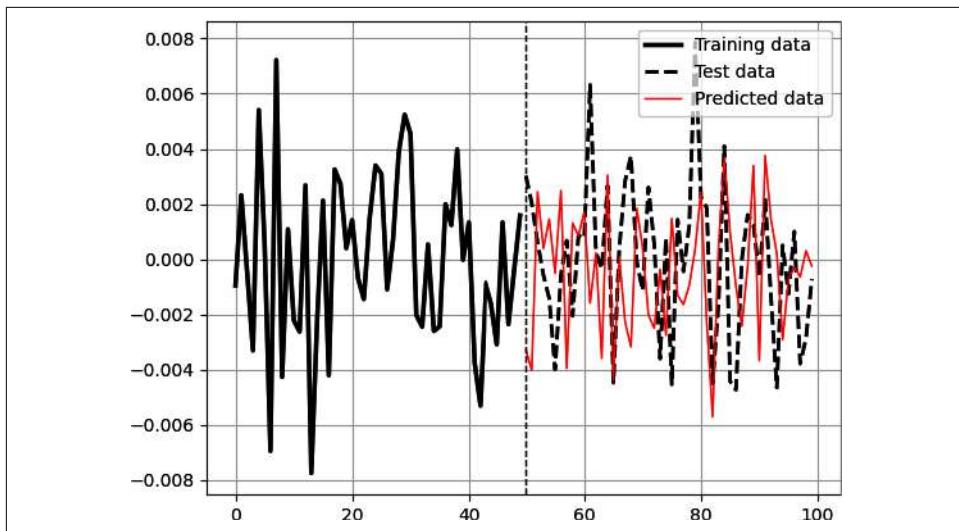


Figure 7-6. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the SGD algorithm.

The model's results are as follows:

```

Accuracy Train = 55.59 %
Accuracy Test = 46.45 %
RMSE Train = 0.007834505
RMSE Test = 0.0059334014
Correlation In-Sample Predicted/Train = 0.235
Correlation Out-of-Sample Predicted/Test = -0.001
Model Bias = 0.95

```

The advantages of SGD are:

- It performs well with large datasets since it updates the model parameters incrementally based on individual or small subsets of training examples.
- It can escape local minima and find better global optima (due to its stochastic nature).
- It can improve generalization by exposing the model to different training examples in each iteration, thereby reducing overfitting.

The disadvantages of SGD are:

- The convergence path can be noisy and exhibit more fluctuations compared to deterministic optimization algorithms. This can result in slower convergence or oscillations around the optimal solution.
- It is impacted by feature scaling, which means it is sensitive to such techniques.

Nearest Neighbors Regression

The *nearest neighbors regression* algorithm, also known as k -nearest neighbors (KNN) regression, is a nonparametric³ algorithm used for regression tasks. It predicts the value of a target variable based on the values of its nearest neighbors in the feature space. The algorithm starts by determining k , which is the number of nearest neighbors to consider when making predictions. This is a hyperparameter that you need to choose based on the problem at hand.



A larger k value provides a smoother prediction, while a smaller k value captures more local variations but may be more prone to noise.

Then the model calculates the distance between the new, unseen data point and all the data points in the training set. The choice of distance metric depends on the nature of the input features. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance. Next, the algorithm selects the k data points with the shortest distances to the query point. These data points are the *nearest neighbors* and will be used to make predictions.

To import the KNN regressor, use the following code:

```
from sklearn.neighbors import KNeighborsRegressor
```

Now let's look at the algorithm's implementation. Fit the model with $k = 10$:

```
# Fitting the model
model = KNeighborsRegressor(n_neighbors = 10)
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

³ A class of statistical methods that do not rely on specific assumptions about the underlying probability distribution.

Figure 7-7 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

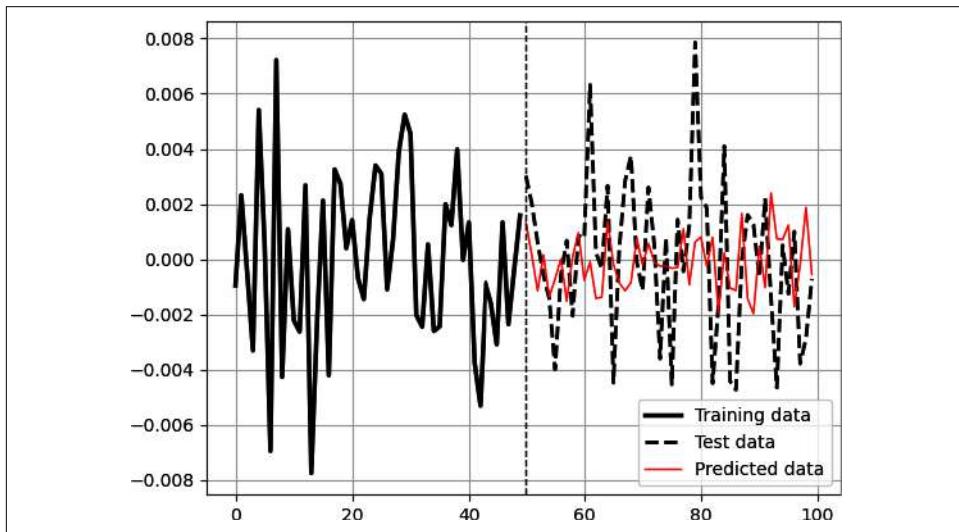


Figure 7-7. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the KNN regression algorithm.

The choice of the number of neighbors in a time series prediction using the KNN regressor depends on several factors, including the characteristics of your dataset and the desired level of accuracy. There is no definitive answer as to how many neighbors to choose, as it is often determined through experimentation and validation. Typically, selecting an appropriate value for the number of neighbors involves a trade-off between bias and variance:

- Small k values are associated with a model that can capture local patterns in the data, but it may also be sensitive to noise or outliers.
- Larger k values are associated with a model that can become more robust to noise or outliers but may overlook local patterns in the data.



If you take the limit as k approaches the size of the dataset, you will get a model that just predicts the class that appears more frequently in the dataset. This is known as the *Bayes error*.

The model's results are as follows:

```
Accuracy Train = 67.69 %
Accuracy Test = 50.77 %
RMSE Train = 0.0069584171
RMSE Test = 0.0054027335
Correlation In-Sample Predicted/Train = 0.599
Correlation Out-of-Sample Predicted/Test = 0.002
Model Bias = 0.76
```



It's essential to consider the temporal aspect of your time series data. If there are clear trends or patterns that span multiple data points, a larger k value might be appropriate to capture those dependencies. However, if the time series exhibits rapid changes or short-term fluctuations, a smaller k value could be more suitable.

The size of your dataset can also influence the choice of k . If you have a small dataset, choosing a smaller value for k might be preferable to avoid overfitting. Conversely, a larger dataset can tolerate a higher value for k .

The advantages of KNN are:

- Its nonlinearity allows it to capture complex patterns in financial data, which can be advantageous for predicting returns series that may exhibit nonlinear behavior.
- It can adapt to changing market conditions or patterns. As the algorithm is instance based, it does not require retraining the model when new data becomes available. This adaptability can be beneficial in the context of financial returns, where market dynamics can change over time.
- It provides intuitive interpretations for predictions. Since the algorithm selects the k nearest neighbors to make predictions, it can be easier to understand and explain compared to more complex algorithms.

The disadvantages of KNN are:

- Its performance can degrade when dealing with high-dimensional data. Financial returns series often involve multiple predictors (such as technical indicators and other correlated returns), and KNN may struggle to find meaningful neighbors in high-dimensional spaces.
- As the dataset grows in size, the computational requirements of KNN can become significant.
- It is sensitive to noisy or outlier data points since the algorithm considers all neighbors equally.

Decision Tree Regression

Decision trees are versatile and intuitive machine learning models. They are graphical representations of a series of decisions or choices based on feature values that lead to different outcomes. Decision trees are structured as a hierarchical flowchart, where each internal node represents a decision based on a feature, each branch represents an outcome of that decision, and each leaf node represents the final prediction or class label.

At the root of the decision tree, consider all the input features and choose the one that best separates the data based on a specific criterion (e.g., the information gain metric discussed in [Chapter 2](#)). Create a decision node associated with the selected feature. Split the data based on the possible values of the chosen feature. Repeat the preceding steps recursively for each subset of data created by the splits, considering the remaining features at each node. Stop the recursion when a stopping criterion is met, such as reaching a maximum depth, reaching a minimum number of samples in a node, or no further improvement in impurity or gain.

To import the decision tree regressor, use the following code:

```
from sklearn.tree import DecisionTreeRegressor
```

Now let's look at the algorithm's implementation:

```
# Fitting the model
model = DecisionTreeRegressor(random_state = 123)
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```



The argument `random_state` is often used to initialize the randomization within algorithms that involve randomness such as initializing weights. This ensures that if you train a model multiple times with the same `random_state`, you'll get the same results, which is important for comparing different algorithms or hyperparameters.

[Figure 7-8](#) shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

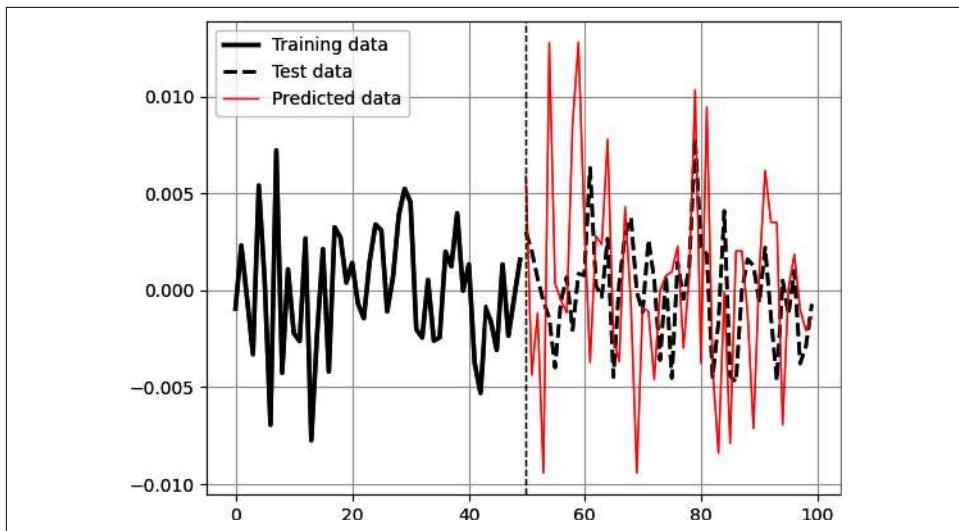


Figure 7-8. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the decision tree regression algorithm.

The model's results are as follows:

```

Accuracy Train = 100.0 %
Accuracy Test = 47.37 %
RMSE Train = 0.0
RMSE Test = 0.007640736
Correlation In-Sample Predicted/Train = 1.0
Correlation Out-of-Sample Predicted/Test = -0.079
Model Bias = 0.94

```

Notice how the accuracy of the training set is extremely high. This is clearly evidence of overfitting (supported by the RMSE of the training data).

The advantages of decision trees are:

- They require minimal data preprocessing and can handle missing values.
- They can capture nonlinear relationships, interactions, and variable importance.

The disadvantages of decision trees are:

- They can be sensitive to small changes in the data and can easily overfit if not properly regularized.
- They may struggle to capture complex relationships that require deeper trees.

The next section presents another breed of machine learning algorithms. These are called *ensemble algorithms*. Decision trees can be combined using ensemble methods to create more robust and accurate models. Random forest, an algorithm seen in the next section, combines multiple decision trees to enhance predictive ability and, especially, to reduce the risk of overfitting.

Random Forest Regression

Random forest is a machine learning algorithm that harnesses the power of multiple decision trees to form a single output (prediction). It is flexible and does not require much tuning. It is also less prone to overfitting due to its ensemble learning technique. *Ensemble learning* refers to the combination of multiple learners (models) to improve the final prediction.

With random forest, the multiple learners are different decision trees that converge toward a single prediction.

Therefore, one of the hyperparameters that can be tuned in random forest algorithms is the number of decision trees. The algorithm uses the bagging method. In the context of random forests, *bagging* refers to the technique of *bootstrap aggregating* that aims to improve the performance and robustness of machine learning models, such as decision trees, by reducing biases. Here's how bagging works within the random forest algorithm:

1. *Bootstrap sampling*: Random forest employs bootstrapping, which means creating multiple subsets of the original training data by sampling with replacement. Each subset has the same size as the original dataset but may contain duplicate instances and exclude some of them. This process is performed independently for each tree in the random forest.
2. *Tree construction and feature selection*: For each bootstrap sample, a decision tree is constructed using a process called *recursive partitioning* where data is split based on features in order to create branches that optimize the separation of the target variables. At each node of the decision tree, a random subset of features is considered for splitting. This helps introduce diversity among the trees in the forest and prevents them from relying too heavily on a single dominant feature.
3. *Ensemble prediction*: Once all the trees are constructed, predictions are made by aggregating the outputs of individual trees. For regression tasks, the predictions are averaged.

To import the random forest regressor, use the following code:

```
from sklearn.ensemble import RandomForestRegressor
```

Now let's look at the algorithm's implementation:

```
# Fitting the model
model = RandomForestRegressor(max_depth = 20, random_state = 123)
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```



The `max_depth` hyperparameter controls the depth of each decision tree in the random forest. A decision tree with a larger depth can capture more intricate patterns in the data, but it also becomes more prone to overfitting, which means it might perform very well on the training data but poorly on unseen data. On the other hand, a shallower tree might not capture all the details of the data but could generalize better to new, unseen data.

Figure 7-9 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

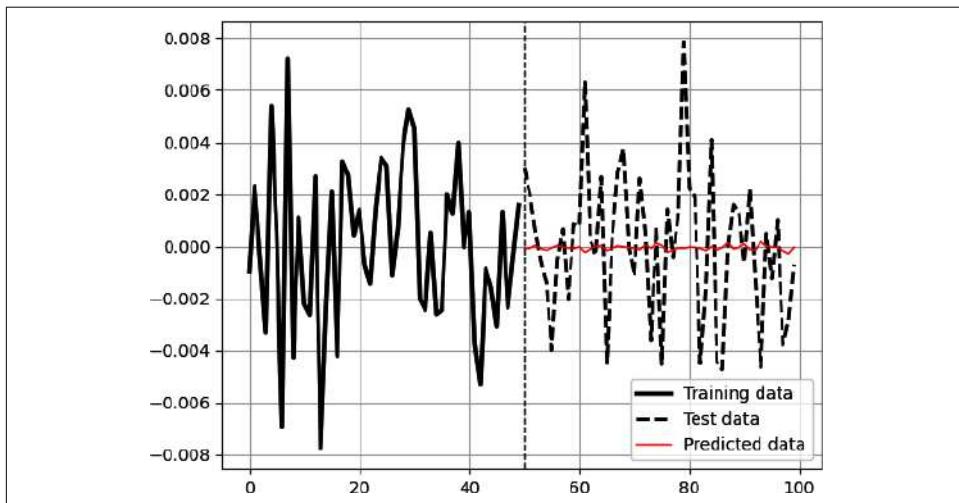


Figure 7-9. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the random forest regression algorithm.

The model's results are as follows:

```
Accuracy Train = 82.72 %
Accuracy Test = 50.15 %
RMSE Train = 0.0058106512
RMSE Test = 0.0053452064
```

```
Correlation In-Sample Predicted/Train =  0.809  
Correlation Out-of-Sample Predicted/Test = -0.049  
Model Bias =  0.63
```

The advantages of random forest regression are as follows:

- It generally has accurate forecasts on data due to its ensemble nature. With financial time series being highly noisy and borderline random, its results need to be optimized nevertheless.
- It exhibits robustness to noise and outliers due to its averaging nature.

The disadvantages of random forest regression are as follows:

- It may be difficult to interpret from time to time. Since it uses an aggregating method, the true and final decision may be lost when a large number of trees is used.
- As the number of trees increases, the computational time of the algorithm takes more time to train, resulting in a slow process.

AdaBoost Regression

Before understanding what AdaBoost is about, let's discuss gradient boosting so that it becomes easier to comprehend the algorithm behind it. *Gradient boosting* is a technique to build models based on the idea of improving *weak learners* (which means models that perform only slightly better than random).

The way to improve these weak learners is to target their weak spots by creating other weak learners that can handle the weak spots. This gave birth to what is known as *Adaptive Boosting*, or *AdaBoost* for short. Hence, in layperson's terms, *boosting* is all about combining weak learners to form better models.

The learners in AdaBoost (which, as discussed, are weak) are single-split decision trees (referred to as *stumps*). They are weighted, with more weight put on instances that are more difficult to classify and less weight put on the rest. At the same time, new learners are incorporated to be trained on the difficult parts, thus creating a more powerful model. Therefore, the difficult instances receive greater weights until they are solved by new weak learners.

Predictions are based on votes from the weak learners. The majority rule is applied in order to maximize accuracy. Gradient boosting therefore can be summarized in three steps:

1. It builds an ensemble of weak predictive models, typically decision trees, in a sequential manner.

- Each subsequent model is built to correct the errors or residuals of the previous models using gradient descent, which adjusts the predictions to minimize overall error.
- The predictions from all the models are combined by taking a weighted average or sum, determined by the learning rate, to produce the final prediction.

To import the AdaBoost regressor, use the following code:

```
from sklearn.ensemble import AdaBoostRegressor
```

Now let's look at the algorithm's implementation:

```
# Fitting the model
model = AdaBoostRegressor(random_state = 123)
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

Figure 7-10 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

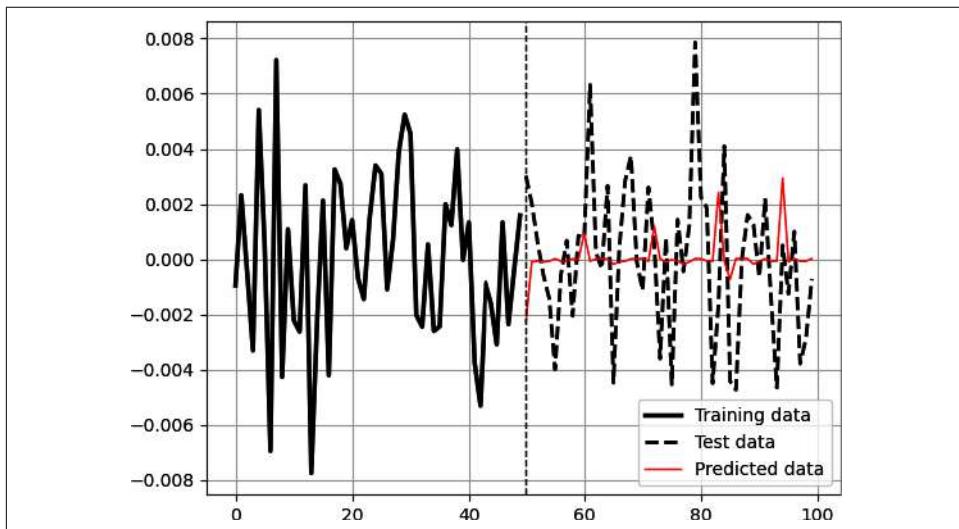


Figure 7-10. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the AdaBoost regression algorithm.

The model's results are as follows:

```
Accuracy Train = 53.27 %
Accuracy Test = 51.7 %
RMSE Train = 0.0070124217
RMSE Test = 0.0053582343
Correlation In-Sample Predicted/Train = 0.461
Correlation Out-of-Sample Predicted/Test = 0.017
Model Bias = 0.72
```

The advantages of AdaBoost are:

- It generally has good accuracy.
- It is easy to comprehend.

The disadvantages of AdaBoost are:

- It is impacted by outliers and sensitive to noise.
- It is slow and not optimized.

XGBoost Regression

XGBoost is a fast and performant gradient-boosted decision tree algorithm. The name may be complicated, but the concept is not hard to understand if you understood gradient boosting from the previous section on AdaBoost. XGBoost stands for *extreme gradient boosting* and was created by Tianqi Chen. Here's how it works:

1. XGBoost starts with a simple base model, usually a decision tree.
2. It defines an objective function that measures the performance of the model.
3. Using gradient descent optimization, it iteratively improves the model's predictions by adjusting the model based on the gradient of the objective function.
4. New decision trees are added to the ensemble to correct errors made by previous models.
5. Regularization techniques, such as learning rate and column subsampling, are employed to enhance performance and prevent fitting issues.
6. The final prediction is obtained by combining the predictions from all the models in the ensemble.

The implementation of XGBoost in Python takes more steps than the previous algorithms. The first step is to `pip install` the required module. Type the following command in the prompt:⁴

```
pip install xgboost
```

To import the XGBoost library, use the following code:

```
from xgboost import XGBRegressor
```

The implementation of the algorithm is as follows:

```
# Fitting the model
model = XGBRegressor(random_state = 123, n_estimators = 16,
                      max_depth = 12)
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```



The argument `n_estimators` is a hyperparameter that determines the number of boosting rounds or trees to be built in the ensemble. As the algorithm combines the predictions of multiple weak learners (individual decision trees) to create a strong predictive model, each boosting round (iteration) adds a new decision tree to the ensemble, and the algorithm learns from the mistakes made by previous trees. The `n_estimators` hyperparameter controls the maximum number of trees that will be added to the ensemble during the training process.

AdaBoost and XGBoost are both boosting algorithms used to enhance the predictive power of weak learners, usually decision trees. AdaBoost focuses on iteratively emphasizing misclassified samples using exponential loss, lacks built-in regularization, and has limited parallelization. In contrast, XGBoost leverages gradient boosting, supports various loss functions, offers regularization, handles missing values, scales better through parallelization, provides comprehensive feature importance, and allows for more extensive hyperparameter tuning.

XGBoost therefore offers more advanced features. It is often preferred for its overall better performance and ability to handle complex tasks. However, the choice between the two depends on the specific problem, dataset, and computational resources available.

⁴ The prompt is a command-line interface that can generally be accessed in the Start menu. It is not the same as the area where you type the Python code that will later be executed.

Figure 7-11 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

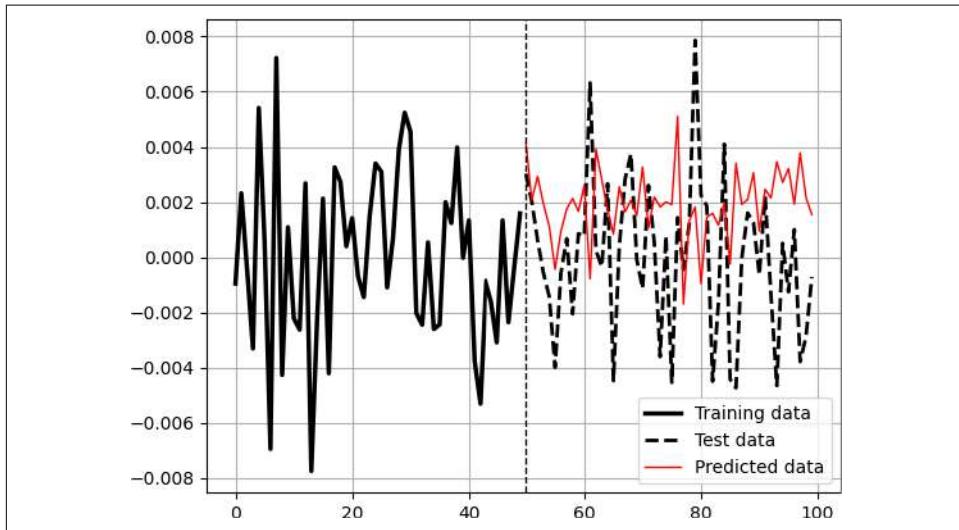


Figure 7-11. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the XGBoost regression algorithm.

The model's results are as follows:

```
Accuracy Train = 75.77 %
Accuracy Test = 53.04 %
RMSE Train = 0.0042354698
RMSE Test = 0.0056622704
Correlation In-Sample Predicted/Train = 0.923
Correlation Out-of-Sample Predicted/Test = 0.05
Model Bias = 6.8
```

Overfitting and Underfitting

Issues will arise in machine-based predictive analytics, and this is completely normal, since *perfection* is an impossible word in the world of data science (and finance). This section covers the most important issue when it comes to predicting data, and that is the *fitting problem*. Overfitting and underfitting are two terms that you must thoroughly understand so that you avoid their consequences when running your models.

Overfitting occurs when a model performs extremely well on the training data but has bad results on the test data. It is a sign that the model has learned not only the details of the in-sample data but also the noise that occurred. Overfitting is generally associated with a high variance and low bias model, but what do those two terms mean?

Bias refers to the difference between the expected value of the model's predictions and the real value of the target variable. A low bias model is one that is complex enough to capture the underlying patterns in the data.

Variance refers to the variability of the model's predictions for different training sets. A high variance model is one that is overly complex and can capture random noise and fluctuations in the training data. This can lead to overfitting, as the model may be fitting the noise in the data.

To prevent overfitting, it's important to strike a balance between bias and variance by selecting a model that is complex enough to capture the underlying patterns in the data but not so complex that it captures random noise and fluctuations in the data. Regularization techniques can also be used to reduce variance and prevent overfitting.

Overfitting occurs for a number of reasons, notably:

Insufficient data

If the training data is not diverse enough, or if there is not enough of it, the model may overfit to the training data.

Overly complex model

If the model is too complex, it may learn the noise in the data rather than the underlying patterns.

Feature overload

If the model is trained on too many features, it may learn irrelevant or noisy features that do not generalize to new data.

Lack of regularization

If the model is not regularized properly, it may overfit to the training data.

Leakage

Leakage occurs when information from the test set is inadvertently included in the training set. This can lead to overfitting as the model is learning from data that it will later see during testing.

A high bias model is one that is overly simplified and cannot capture the true underlying patterns in the data. This can lead to underfitting. Similarly, a low variance model is one that is not affected much by small changes in the training data and can generalize well to new, unseen data.

Underfitting occurs for a number of reasons, notably:

Insufficient model complexity

If the model used is too simple to capture the underlying patterns in the data, it may result in underfitting. For example, a linear regression model might not be able to capture the nonlinear relationship between the features and the target variable.

Insufficient training

If the model is not trained for long enough, or with enough data, it may not be able to capture the underlying patterns in the data.

Over-regularization

Regularization is a technique used to prevent overfitting, but if it's used excessively, it can lead to underfitting.

Poor feature selection

If the features selected for the model are not informative or relevant, the model may underfit.

Figure 7-12 shows a comparison between the different fits of a model to the data. An underfit model fails to capture the real relationship from the start, thus it is bad at predicting the past values and the future ones as well. A well-fit model captures the general tendency of the data. It is not an exact or a perfect model but one that generally has satisfactory predictions across the time period. An overfit model captures every detail of the past, even if it's noise or random dislocations. The danger of an overfit model is that it inhibits a false promise of the future.

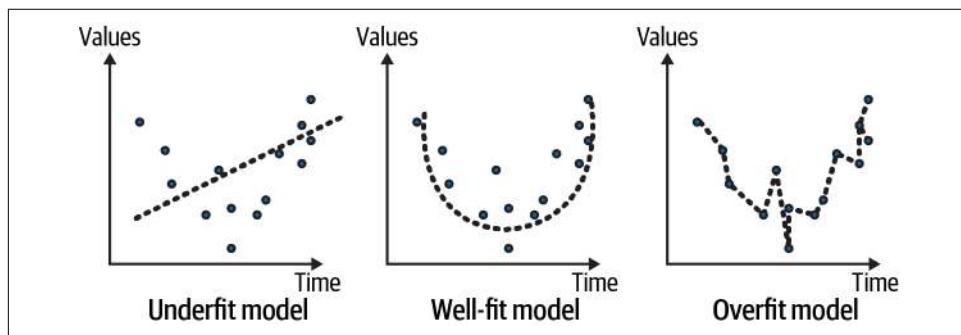


Figure 7-12. Different fitting situations.

Therefore, when building machine learning models for time series prediction, you have to make sure you do not tune the parameters to perfectly fit the past values. To reduce fitting biases, make sure you incorporate the following best practices in your backtests:

Increase training data

Collecting more training data helps to capture a broader range of patterns and variations in the data, reducing the chances of overfitting.

Feature selection

Carefully select relevant and informative features for your model. Removing irrelevant or redundant features reduces noise and complexity in the data, making it easier for the model to generalize well to unseen examples.

Regularization techniques

Regularization methods explicitly control the complexity of the model to prevent overfitting.

Hyperparameter tuning

Optimize the hyperparameters of your model to find the best configuration. Hyperparameters control the behavior and complexity of the model.

Ensemble methods

Employ ensemble methods, such as random forests, to combine predictions from multiple models. Ensemble methods can reduce overfitting by aggregating the predictions of multiple models, smoothing out individual model biases, and improving generalization.

Regular model evaluation

Regularly evaluate your model's performance on unseen data or a dedicated validation set. This helps monitor the model's generalization ability and detect any signs of overfitting or degradation in performance.

Summary

By properly understanding where machine learning algorithms come from, it becomes simpler to interpret them and understand their limitations. This chapter gave you the required knowledge (theory and practice) to build time series models using a few known machine learning algorithms in the hopes of forecasting values using past values.

What you must imperatively know is that past values are not necessarily indicative of future outcomes. Backtests are always biased somehow since a lot of tweaking is needed to tune the results, which may cause overfitting. Patterns do occur, but their results are not necessarily the same. Machine learning for financial time series prediction is constantly evolving, and most of the algorithms (in the raw form and with their basic inputs) are not very predictive, but with proper combination and the addition of risk management tools and filters, you may have a sustainable algorithm that adds value to your whole framework.

Deep Learning for Time Series Prediction I

Deep learning is a slightly more complex and more detailed field than machine learning. Machine learning and deep learning both fall under the umbrella of data science. As you will see, deep learning is mostly about neural networks, a highly sophisticated and powerful algorithm that has enjoyed a lot of coverage and hype, and for good reason: it is very powerful and able to catch highly complex nonlinear relationships between different variables.

The aim of this chapter is to explain the functioning of neural networks before using them to predict financial time series in Python, just like you saw in [Chapter 7](#).

A Walk Through Neural Networks

Artificial neural networks (ANNs) have their roots in the study of neurology, where researchers sought to comprehend how the human brain and its intricate network of interconnected neurons functioned. ANNs are designed to produce computational representations of biological neural network behavior.

ANNs have been around since the 1940s, when academics first started looking into ways to build computational models based on the human brain. Logician Walter Pitts and neurophysiologist Warren McCulloch were among the early pioneers in this subject. They published the idea of a computational model based on simplified artificial neurons in a paper.¹

The development of artificial neural networks gained further momentum in the 1950s and 1960s when researchers like Frank Rosenblatt worked on the *perceptron*, a

¹ W. S. McCulloch and W. Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity” *Bulletin of Mathematical Biophysics* 5 (1943): 115–33.

type of artificial neuron that could learn from its inputs. Rosenblatt's work paved the way for the development of single-layer neural networks capable of pattern recognition tasks.

With the creation of multilayer neural networks, also known as *deep neural networks*, and the introduction of more potent algorithms, artificial neural networks made significant strides in the 1980s and 1990s. This innovation made it possible for neural networks to learn hierarchical data representations, which enhanced their performance on challenging tasks. Although multiple researchers contributed to the development and advancement of artificial neural networks, one influential figure is Geoffrey Hinton. Hinton, along with his collaborators, made significant contributions to the field by developing new learning algorithms and architectures for neural networks. His work on deep learning has been instrumental in the recent resurgence and success of artificial neural networks.

An ANN consists of interconnected nodes, called artificial neurons, organized into layers. The layers are typically divided into three types:

Input layer

The input layer receives input data, which could be numerical, categorical, or even raw sensory data. Input layers are explanatory variables that are supposed to be predictive in nature.

Hidden layers

The hidden layers (one or more) process the input data through their interconnected neurons. Each neuron in a layer receives inputs, performs a computation (discussed later), and passes the output to the next layer.

Output layer

The output layer produces the final result or prediction based on the processed information from the hidden layers. The number of neurons in the output layer depends on the type of problem the network is designed to solve.

Figure 8-1 shows an illustration of an artificial neural network where the information flows from left to right. It begins with the two inputs being connected to the four hidden layers where calculation is done before outputting a weighted prediction in the output layer.

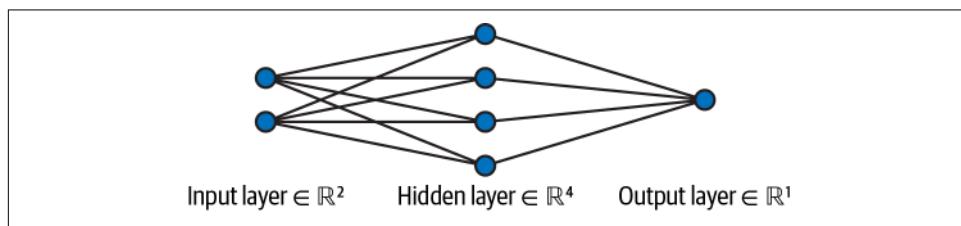


Figure 8-1. A simple illustration of an artificial neural network.

Each neuron in the ANN performs two main operations:

1. The neuron receives inputs from the previous layer or directly from the input data. Each input is multiplied by a weight value, which represents the strength or importance of that connection. The weighted inputs are then summed together.
2. After the weighted sum, an activation function (discussed in the next section) is applied to introduce nonlinearity into the output of the neuron. The activation function determines the neuron's output value based on the summed inputs.

During the training process, the ANN adjusts the weights of its connections to improve its performance. This is typically done through an iterative optimization algorithm, such as gradient descent, where the network's performance is evaluated using a defined loss function. The algorithm computes the gradient of the loss function with respect to the network's weights, allowing the weights to be updated in a way that minimizes the error.

ANNs have the ability to learn and generalize from data, making them suitable for tasks like pattern recognition and regression. With the advancements in deep learning, ANNs with multiple hidden layers have shown exceptional performance on complex tasks, leveraging their ability to learn hierarchical representations and capture intricate patterns in the data.



It is worth noting that the process from inputs to outputs is referred to as *forward propagation*.

Activation Functions

Activation functions in neural networks introduce nonlinearity to the output of a neuron, allowing neural networks to model complex relationships and learn from nonlinear data. They determine the output of a neuron based on the weighted sum of its inputs. Let's discuss these activation functions in detail.

The *sigmoid activation function* maps the input to a range between 0 and 1, making it suitable for binary classification problems or as a smooth approximation of a step function. The mathematical representation of the function is as follows:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Figure 8-2 shows the sigmoid function.

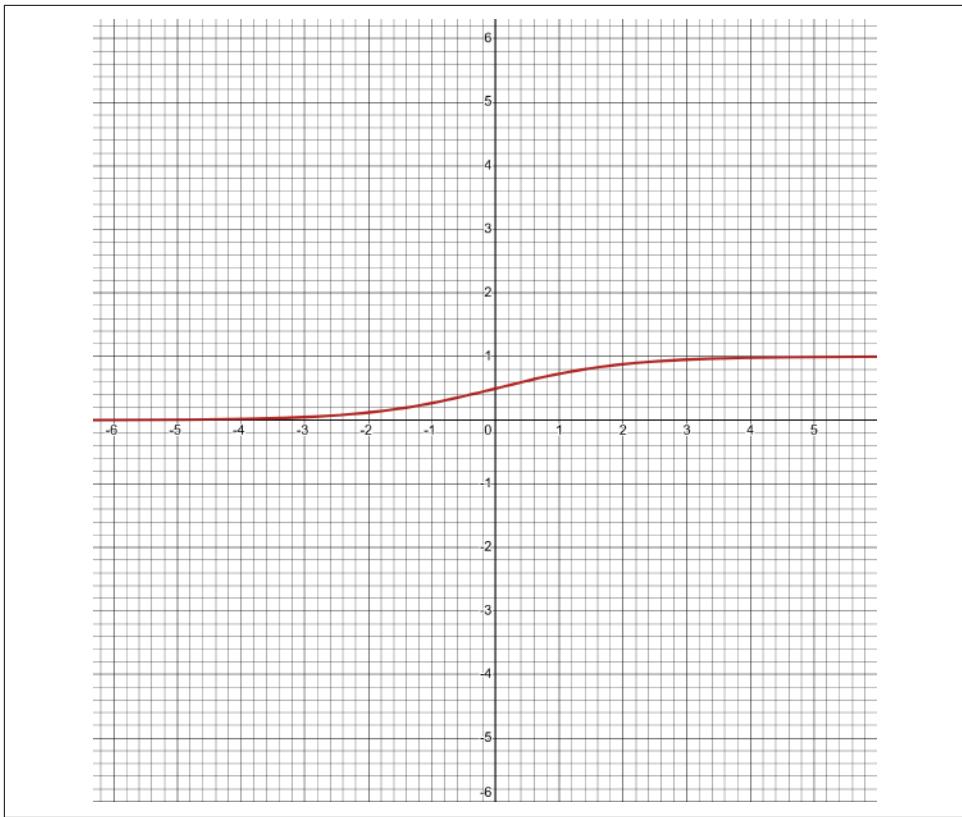


Figure 8-2. Graph of the sigmoid function.

Among the advantages of the sigmoid activation function are the following:

- It is a smooth as well as differentiable function that facilitates gradient-based optimization algorithms.
- It squashes the input to a bounded range, which can be interpreted as a probability or confidence level.

However, it has its limitations as well:

- It suffers from the *vanishing gradient problem*, where gradients become very small for extreme input values. This can hinder the learning process.
- Outputs are not zero centered, making it less suitable for certain situations, such as optimizing weights using symmetric update rules like the gradient descent.

The next activation function is the *hyperbolic tangent function* (\tanh), which you saw in [Chapter 4](#). The mathematical representation of the function is as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Among the advantages of the hyperbolic tangent function are the following:

- It is similar to the sigmoid function but is zero centered, which helps alleviate the issue of asymmetric updates in weight optimization.
- Its nonlinearity can capture a wider range of data variations compared to the sigmoid function.

The following are among its limitations:

- It suffers from the vanishing gradient problem, particularly in deep networks.
- Outputs are still susceptible to saturation at the extremes, resulting in gradients close to zero.

[Figure 8-3](#) shows the hyperbolic tangent function.

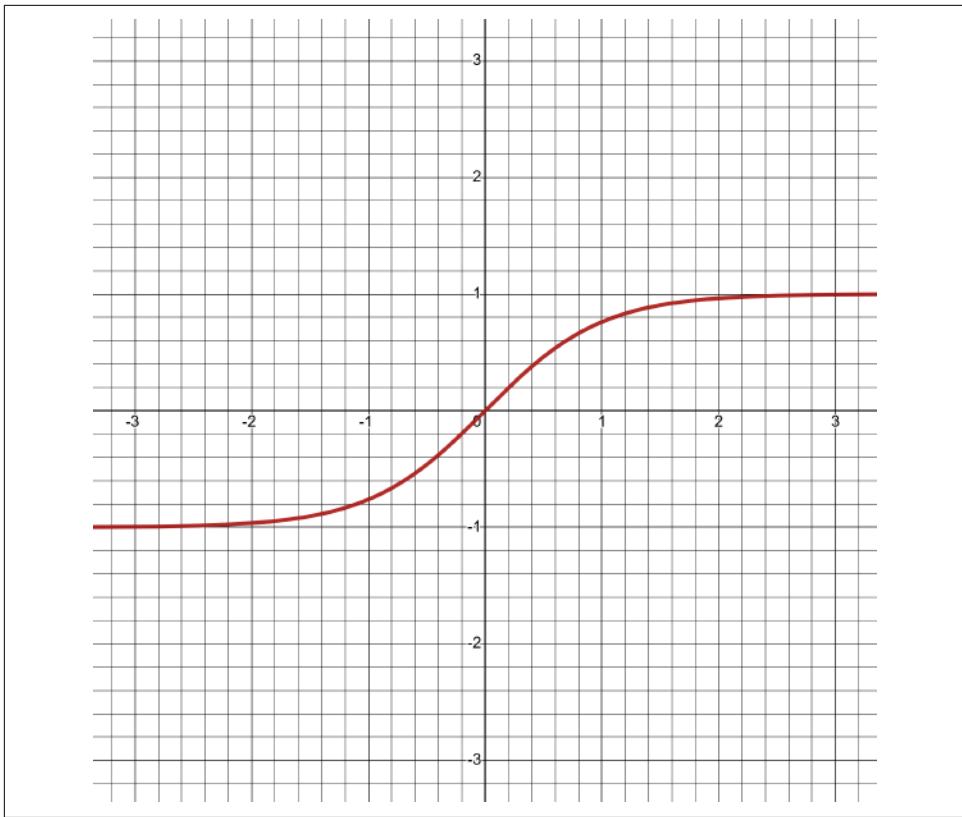


Figure 8-3. Graph of the hyperbolic tangent function.

The next function is called the *ReLU activation function*. ReLU stands for *rectified linear unit*. This function sets negative values to zero and keeps the positive values unchanged. It is efficient and helps avoid the vanishing gradient problem. The mathematical representation of the function is as follows:

$$f(x) = \max(0, x)$$

Among the advantages of the ReLU function are the following:

- It is simple to implement, as it only involves taking the maximum of 0 and the input value. The simplicity of ReLU leads to faster computation and training compared to more complex activation functions.

- It helps mitigate the vanishing gradient problem that can occur during deep neural network training. The derivative of ReLU is either 0 or 1, which means that the gradients can flow more freely and avoid becoming exponentially small as the network gets deeper.

Among the limitations of the function are the following:

- It outputs 0 for negative input values, which can lead to information loss. In some cases, it may be beneficial to have activation functions that can produce negative outputs as well.
- It is not a smooth function, because its derivative is discontinuous at 0. This can cause optimization difficulties in certain scenarios.

Figure 8-4 shows the ReLU function.

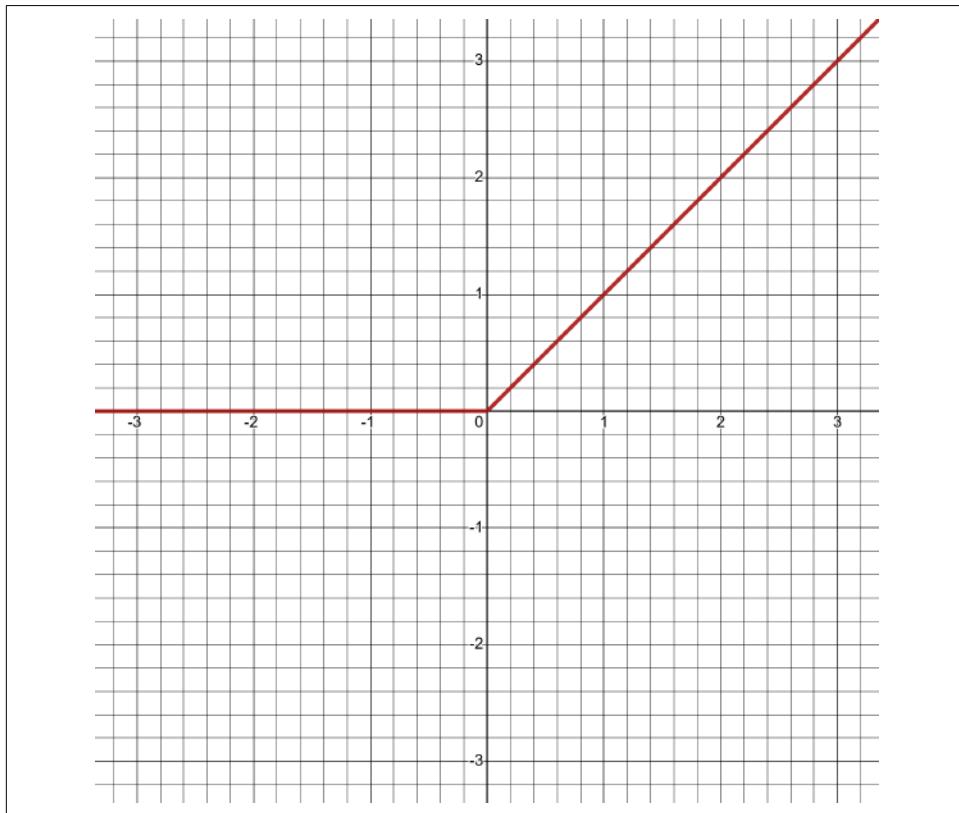


Figure 8-4. Graph of the ReLU function.

The final activation function to discuss is the *leaky ReLU activation function*. This activation function is an extension of the ReLU function that introduces a small slope for negative inputs. The mathematical representation of the function is as follows:

$$f(x) = \max(0.01x, x)$$

Leaky ReLU addresses the dead neuron problem in ReLU and allows some activation for negative inputs, which can help with the flow of gradients during training.

Among the advantages of the leaky ReLU function are the following:

- It overcomes the issue of dead neurons that can occur with ReLU. By introducing a small slope for negative inputs, leaky ReLU ensures that even if a neuron is not activated, it can still contribute to the gradient flow during training.
- It is a continuous function, even at negative input values. The nonzero slope for negative inputs allows the activation function to have a defined derivative throughout its input range.

The following are among the limitations of the function:

- The slope of the leaky part is a hyperparameter that needs to be set manually. It requires careful tuning to strike a balance between avoiding dead neurons and preventing too much leakage that may hinder the nonlinearity of the activation function.
- Although leaky ReLU provides a nonzero response for negative inputs, it does not provide the same level of negative activation as some other activation functions, such as the hyperbolic tangent (tanh) and sigmoid. In scenarios where a strong negative activation response is desired, other activation functions might be more suitable.

Figure 8-5 shows the leaky ReLU function.

Your choice of activation function depends on the nature of the problem, the architecture of the network, and the desired behavior of the neurons in the network.

Activation functions typically take the weighted sum of inputs to a neuron and apply a nonlinear transformation to it. The transformed value is then passed on as the output of the neuron to the next layer of the network. The specific form and behavior of activation functions can vary, but their overall purpose is to introduce nonlinearities that allow the network to learn complex patterns and relationships in the data.

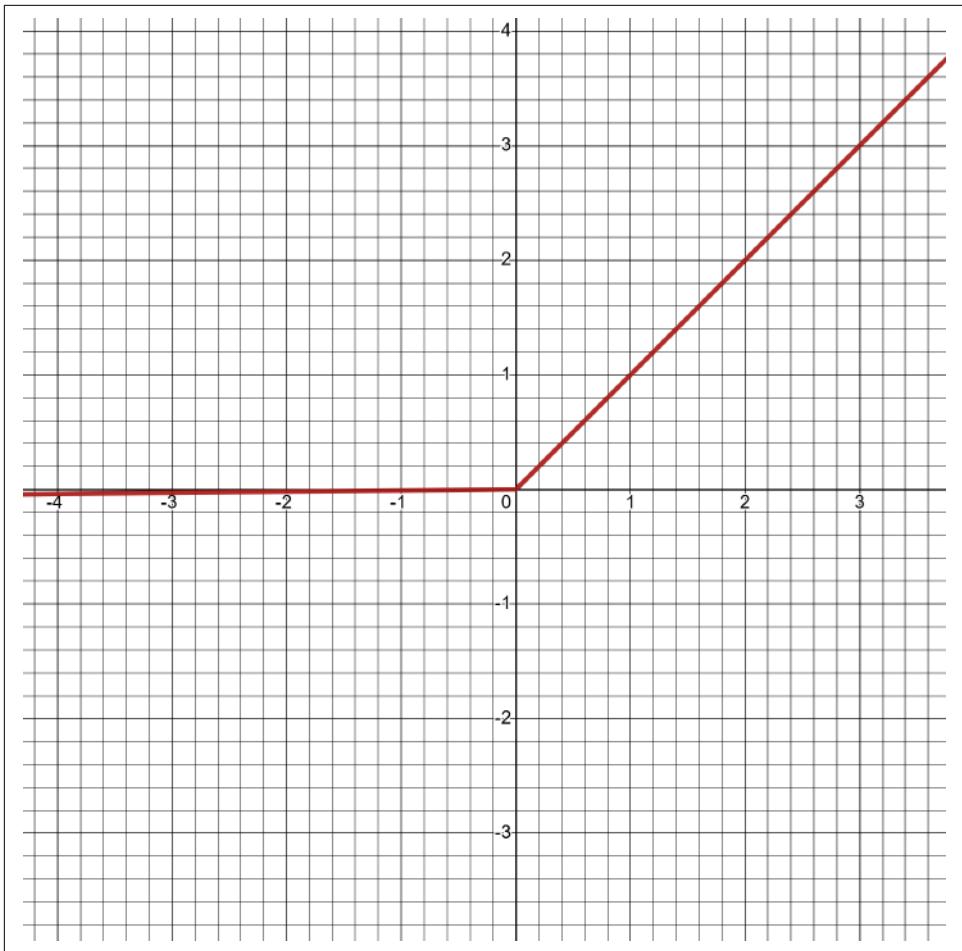


Figure 8-5. Graph of the leaky ReLU function.

To sum up, activation functions play a crucial role in ANNs by introducing nonlinearity into the network's computations. They are applied to the outputs of individual neurons or intermediate layers and help determine whether a neuron should be activated or not based on the input it receives. Without activation functions, the network would only be able to learn linear relationships between the input and output. However, most real-world problems (especially financial time series) involve complex, nonlinear relationships, so activation functions are essential for enabling neural networks to learn and represent such relationships effectively.

Backpropagation

Backpropagation is a fundamental algorithm used to train neural networks. It allows the network to update its weights in a way that minimizes the difference between the predicted output and the desired output.



Backpropagation is a shortened term for *backward propagation of errors*.

Training neural networks involves the following steps:

1. Randomly initialize the weights and biases of the neural network. This allows you to have a first step when you do not have initial information.
2. Perform *forward propagation*, a technique to calculate the predicted outputs of the network for a given input. As a reminder, this step involves calculating the weighted sum of inputs for each neuron, applying the activation function to the weighted sum, passing the value to the next layer (if it's not the last), and continuing the process until reaching the output layer (prediction).
3. Compare the predicted output with the actual output (test data) and calculate the loss, which represents the difference between them. The choice of the loss function (e.g., MAE or MSE) depends on the specific problem being solved.
4. Perform backpropagation to calculate the gradients of the loss with respect to the weights and biases. In this step, the algorithm will start from the output layer (the last layer) and go backward. It will compute the gradient of the loss with respect to the output of each neuron in the current layer. Then it will calculate the gradient of the loss with respect to the weighted sum of inputs for each neuron in the current layer by applying the chain rule. After that, it will compute the gradient of the loss with respect to the weights and biases of each neuron in the current layer using the gradients from the previous steps. These steps are repeated until the gradients are calculated for all layers.
5. Update the weights and biases of the network by using the calculated gradients and a chosen optimization algorithm run on a specific number of batches of data, which are controlled by the hyperparameter (referred to as the batch size). Updating the weights is done by subtracting the product of the learning rate and the gradient of the weights. Adjusting the biases is done by subtracting the product of the learning rate and the gradient of the biases. Repeat the preceding steps until the weights and biases are updated for all layers.
6. The algorithm then repeats steps 2–5 for a specified number of epochs or until a convergence criterion is met. An *epoch* represents one complete pass through the

entire training dataset (the whole process entails passing through the training dataset multiple times ideally).

- Once the training is completed, evaluate the performance of the trained neural network on a separate validation or test dataset.



The *learning rate* is a hyperparameter that determines the step size at which a neural network's weights are updated during the training process. It controls how quickly or slowly the model learns from the data it's being trained on.

The *batch size* is a hyperparameter that determines the number of samples processed before updating the model's weights during each iteration of the training process. In other words, it specifies how many training examples are used at a time to calculate the gradients and update the weights.

Choosing an appropriate batch size is essential for efficient training and can impact the convergence speed and memory requirements. There is no one-size-fits-all answer to the ideal batch size, as it depends on various factors, such as the dataset size, available computational resources, and the complexity of the model.

Commonly used batch sizes for training MLPs range from small values (such as 16, 32, or 64) to larger ones (such as 128, 256, or even larger). Smaller batch sizes can offer more frequent weight updates and may help the model converge more quickly, especially when the dataset is large or has a lot of variations. However, smaller batch sizes may also introduce more noise and slower convergence due to frequent updates with less accurate gradients. On the other hand, larger batch sizes can provide more stable gradients and better utilization of parallel processing capabilities, leading to faster training on modern hardware. However, they might require more memory, and the updates are less frequent, which could slow down convergence or make the training process less robust.

As a general rule of thumb, you can start with a moderate batch size like 32 and experiment with different values to find the best trade-off between convergence speed and computational efficiency for your specific MLP model and dataset.

The backpropagation algorithm leverages the chain rule (refer to [Chapter 4](#) for more information on calculus) to calculate the gradients by propagating the errors backward through the network.

By iteratively adjusting the weights based on the error propagated backward through the network, backpropagation enables the network to learn and improve its predictions over time. Backpropagation is a key algorithm in training neural networks and has contributed to significant advancements in various fields.

Optimization Algorithms

In neural networks, optimization algorithms, also known as *optimizers*, are used to update the parameters (weights and biases) of the network during the training process. These algorithms aim to minimize the loss function and find the optimal values for the parameters that result in the best performance of the network. There are several types of optimizers:

Gradient descent (GD)

Gradient descent is the most fundamental optimization algorithm. It updates the network's weights and biases in the direction opposite to the gradient of the loss function with respect to the parameters. It adjusts the parameters by taking steps proportional to the negative of the gradient, multiplied by a learning rate.

Stochastic gradient descent (SGD)

SGD is a variant of gradient descent that randomly selects a single training example or a mini batch of examples to compute the gradient and update the parameters. It provides a computationally efficient approach and introduces noise in the training process, which can help escape local optima.

Adaptive moment estimation (Adam)

Adam is an adaptive optimization algorithm that computes adaptive learning rates for each parameter based on estimates of the first and second moments of the gradients. Adam is widely used due to its effectiveness and efficiency in various applications.

Root mean square propagation (RMSprop)

The purpose of RMSprop is to address some of the limitations of the standard gradient descent algorithm, such as slow convergence and oscillations in different directions. RMSprop adjusts the learning rate for each parameter based on the average of the recent squared gradients. It calculates an exponentially weighted moving average of the squared gradients over time.

Each optimizer has its own characteristics, advantages, and limitations, and their performance can vary depending on the dataset and the network architecture. Experimentation and tuning are often necessary to determine the best optimizer for a specific task.

Regularization Techniques

Regularization techniques in neural networks are methods used to prevent overfitting, which can lead to poor performance and reduced ability of the model to make accurate predictions on new examples. Regularization techniques help to control the complexity of a neural network and improve its ability to generalize to unseen data.

Dropout is a regularization technique commonly used in neural networks to prevent overfitting (refer to [Chapter 7](#) for detailed information on overfitting). It involves randomly omitting (dropping) a fraction of the neurons during training by setting their outputs to zero. This temporarily removes the neurons and their corresponding connections from the network, forcing the remaining neurons to learn more robust and independent representations.

The key idea behind dropout is that it acts as a form of model averaging or ensemble learning. By randomly dropping out neurons, the network becomes less reliant on specific neurons or connections and learns more robust features. Dropout also helps prevent co-adaptation, where certain neurons rely heavily on others, reducing their individual learning capability. As a result, dropout can improve the network's generalization ability and reduce overfitting.

Early stopping is a technique that also prevents overfitting by monitoring the model's performance on a validation set during training. It works by stopping the training process when the model's performance on the validation set starts to deteriorate. The idea behind early stopping is that as the model continues to train, it may start to overfit the training data, causing a decrease in performance on unseen data.

The training process is typically divided into epochs, where each epoch represents a complete pass over the training data. During training, the model's performance on the validation set is evaluated after each epoch. If the validation loss or a chosen metric starts to worsen consistently for a certain number of epochs, training is stopped, and the model's parameters from the epoch with the best performance are used as the final model.

Early stopping helps prevent overfitting by finding the optimal point at which the model has learned the most useful patterns without memorizing noise or irrelevant details from the training data. Both dropout and early stopping are key regularization techniques that help prevent overfitting and help stabilize the model.

Multilayer Perceptrons

A *multilayer perceptron* (MLP) is a type of ANN that consists of multiple layers of artificial neurons, or nodes, arranged in a sequential manner. It is a *feedforward neural network*, meaning that information flows through the network in one direction, from the input layer to the output layer, without any loops or feedback connections (you will learn more about this later in [“Recurrent Neural Networks” on page 232](#)).

The basic building block of an MLP is a *perceptron*, an artificial neuron that takes multiple inputs, applies weights to those inputs, performs a weighted sum, and passes the result through an activation function to produce an output (basically, the neuron that you have seen already). An MLP contains multiple perceptrons organized in

layers. It typically consists of an input layer, one or more hidden layers (the more layers, the deeper the learning process up to a certain point), and an output layer.



The term *perceptron* is sometimes used more broadly to refer to a single-layer neural network based on a perceptron-like architecture. In this context, the term *perceptron* can be used interchangeably with *neural network* or *single-layer perceptron*.

As a reminder, the input layer receives the raw input data, such as features from a dataset (e.g., the stationary values of a moving average). The hidden layers, which are intermediate layers between the input and output layers, perform complex transformations on the input data. Each neuron in a hidden layer takes inputs from all neurons in the previous layer, applies weights, performs the weighted sum, and passes the result through an activation function. The output layer produces the final output of the network.

MLPs are trained using backpropagation, which adjusts the weights of the neurons in the network to minimize the difference between the predicted output and the desired output. They are known for their ability to learn complex, nonlinear relationships in data, making them suitable for a wide range of tasks, including pattern recognition. [Figure 8-6](#) shows an example of a deep MLP architecture.

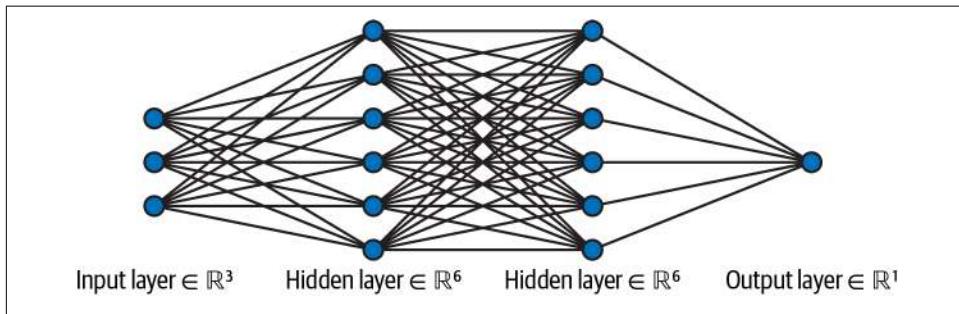


Figure 8-6. A simple illustration of an MLP with two hidden layers.

At this stage, you should understand that deep learning is basically neural networks with many hidden layers that add to the complexity of the learning process.



It is important to download *master_function.py* from this book's [GitHub repository](#) to access the functions seen in this book. After downloading it, you must set your Python's interpreter directory as the path where *master_function.py* is stored.

The aim of this section is to create an MLP to forecast daily S&P 500 returns. Import the required libraries:

```
from keras.models import Sequential
from keras.layers import Dense
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Now import the historical data and transform it:

```
# Set the start and end dates for the data
start_date = '1990-01-01'
end_date = '2023-06-01'
# Fetch S&P 500 price data
data = np.array((pdr.get_data_fred('SP500', start = start_date,
                                    end = end_date)).dropna())
# Difference the data and make it stationary
data = np.diff(data[:, 0])
```

Set the hyperparameters for the model:

```
num_lags = 100
train_test_split = 0.80
num_neurons_in_hidden_layers = 20
num_epochs = 500
batch_size = 16
```

Use the data preprocessing function to create the four required arrays:

```
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data, num_lags,
                                                       train_test_split)
```

The following code block shows how to build the MLP architecture in *keras*. Make sure you understand the notes in the code:

```
# Designing the architecture of the model
model = Sequential()
# First hidden layer with ReLU as activation function
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
                activation = 'relu'))
# Second hidden layer with ReLU as activation function
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(1))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, np.reshape(y_train, (-1, 1)), epochs = num_epochs,
```

```

batch_size = batch_size)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))

```



When creating a `Dense` layer, you need to specify the `input_dim` parameter in the first layer of your neural network. For subsequent `Dense` layers, the `input_dim` is automatically inferred from the previous layer's output.

Let's plot the results and analyze the performance:

```

Accuracy Train = 92.4 %
Accuracy Test = 54.85 %
RMSE Train = 4.3602984254
RMSE Test = 75.7542774467
Correlation In-Sample Predicted/Train = 0.989
Correlation Out-of-Sample Predicted/Test = 0.044
Model Bias = 1.03

```

Figure 8-7 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

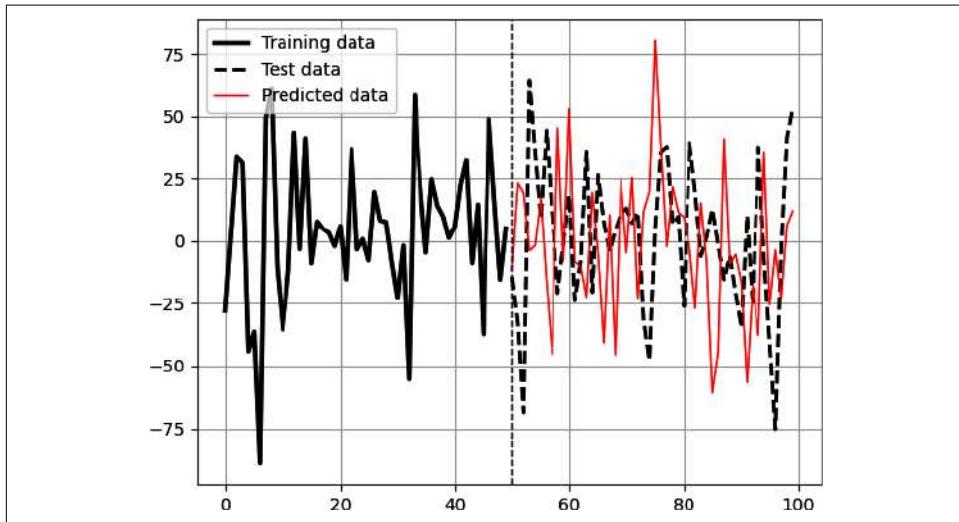


Figure 8-7. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the MLP regression algorithm.

The results are extremely volatile when changing the hyperparameters. This is why using sophisticated models on complex data requires a lot of tweaks and optimizations. Consider the following improvements to enhance the results of the model:

- Select relevant features (inputs) that capture the underlying patterns and characteristics of the financial time series. This can involve calculating technical indicators (e.g., moving averages and the RSI) or deriving other meaningful variables from the data.
- Review the architecture of the model. Consider increasing the number of layers or neurons to provide the model with more capacity to learn complex patterns. Experiment with different activation functions and regularization techniques such as dropout and early stopping (see [Chapter 9](#) for an application of regularization techniques).
- Fine-tune the hyperparameters of your MLP model. Parameters like the batch size and the number of epochs can significantly impact the model's ability to converge and generalize.
- Combine multiple MLP models into an ensemble. This can involve training several models with different initializations or using different subsets of the data. Aggregating their predictions can lead to better results than using a single model.

As the model trains, the loss function should decrease due to the learning process. This can be seen using the following code (to be run after compiling the model):

```
import tensorflow as tf
losses = []
epochs = []
class LossCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs = None):
        losses.append(logs['loss'])
        epochs.append(epoch + 1)
        plt.clf()
        plt.plot(epochs, losses, marker = 'o')
        plt.title('Loss Curve')
        plt.xlabel('Epoch')
        plt.ylabel('Loss Value')
        plt.grid(True)
        plt.pause(0.01)
model.fit(x_train, np.reshape(y_train, (-1, 1)), epochs = 100,
          verbose = 0, callbacks = [LossCallback()])
plt.show()
```

The previous code block plots the loss at the end of every epoch, thus creating a dynamic loss curve visualized in real time. Notice how it falls until reaching a plateau where it struggles to decrease. [Figure 8-8](#) shows the decreasing loss function across epochs.

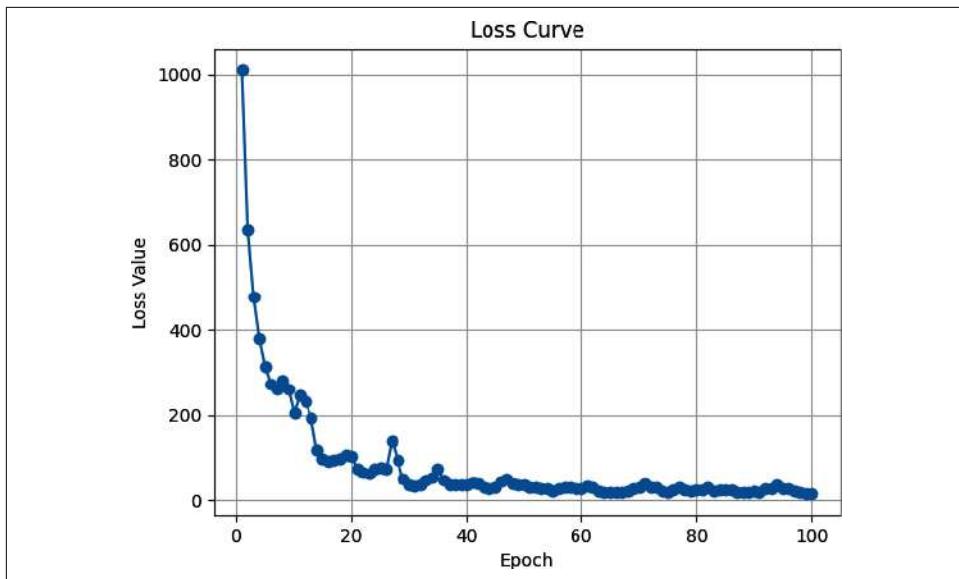


Figure 8-8. Loss value across epochs.

Recurrent Neural Networks

A *recurrent neural network* (RNN) is a type of artificial neural network that is designed to process sequential data or data with temporal dependencies. Unlike feed-forward neural networks, which process data in a single pass from input to output, RNNs maintain internal memory or hidden states to capture information from previous inputs and utilize it in the processing of subsequent inputs.

The key feature of an RNN is the presence of *recurrent connections*, which create a loop in the network. This loop allows the network to persist information across time steps, making it well suited for tasks that involve sequential or time-dependent data.

At each time step, an RNN takes an input vector and combines it with the previous hidden state. It then applies activation functions to compute the new hidden state and produces an output. This process is repeated for each time step, with the hidden state being updated and passed along as information flows through the network.

The recurrent connections enable RNNs to capture dependencies and patterns in sequential data. They can model the context and temporal dynamics of the data, making them useful in time series prediction.

However, traditional RNNs suffer from the vanishing gradient problem, where the gradients that are backpropagated through the recurrent connections can become very small or very large, leading to difficulties in training the network. The vanishing

gradient problem is resolved in the next section with an enhanced type of neural network. For now, let's focus on RNNs and their specificities.

Figure 8-9 shows an example of an RNN architecture.

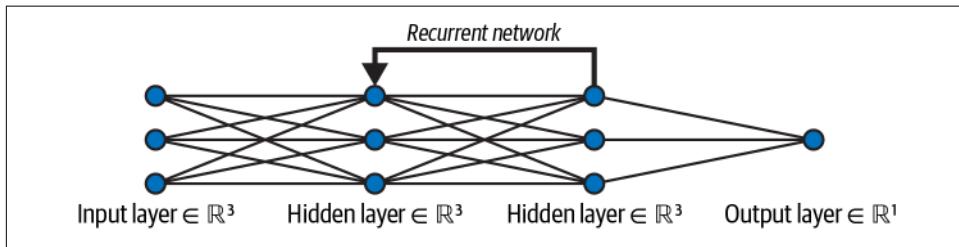


Figure 8-9. A simple illustration of an RNN with two hidden layers.

Let's deploy an RNN algorithm to forecast S&P 500 daily returns. As usual, import the required libraries:

```
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Now set the hyperparameters of the model:

```
num_lags = 100
train_test_split = 0.80
num_neurons_in_hidden_layers = 20
num_epochs = 500
batch_size = 16
```

The following code block shows how to build the RNN architecture in *keras*:

```
# Designing the architecture of the model
model = Sequential()
# First hidden layer
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
               activation = 'relu'))
# Second hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(1))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, np.reshape(y_train, (-1, 1)), epochs = num_epochs,
          batch_size = batch_size)
```

```

# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))

```

Let's plot the results and analyze the performance:

```

Accuracy Train = 67.16 %
Accuracy Test = 52.11 %
RMSE Train = 22.7704952044
RMSE Test = 60.3443059267
Correlation In-Sample Predicted/Train = 0.642
Correlation Out-of-Sample Predicted/Test = -0.022
Model Bias = 2.18

```

Figure 8-10 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

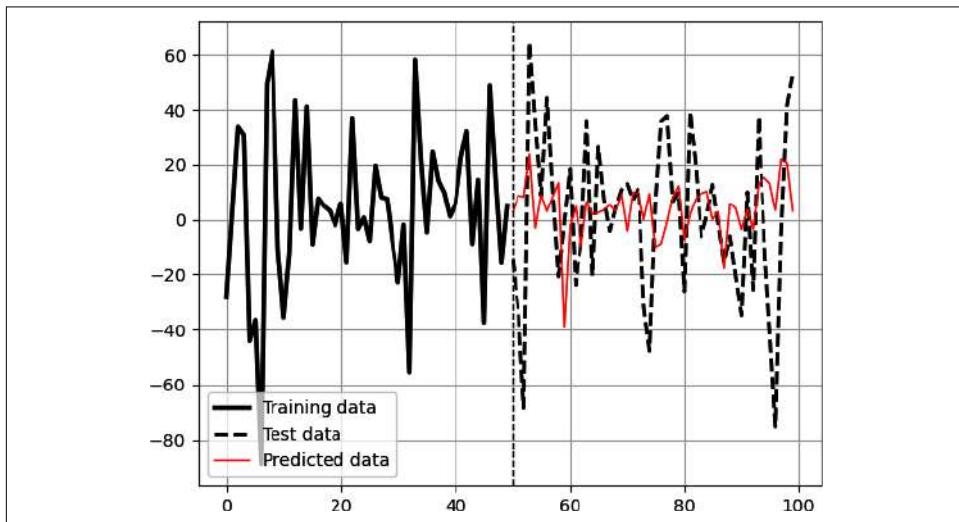


Figure 8-10. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the RNN regression algorithm.



A good task for you to do is to create an optimization function that loops around different hyperparameters and selects the best ones or averages the best ones. This way, you may be able to obtain a robust model based on the ensembling technique. You can also backtest different markets and different time horizons. Note that these techniques are valid not only for financial time series, but for all types of time series.

In summary, RNNs are neural networks that can process sequential data by maintaining internal memory and capturing temporal dependencies. They are powerful models for tasks involving time series or sequential data. As a reminder, stationarity is an essential property for successful time series forecasting. A stationary time series exhibits constant mean, variance, and autocovariance over time. RNNs (among other deep learning models) assume that the underlying time series is stationary, which means the statistical properties of the data do not change over time. If the time series is nonstationary, it may contain trends, seasonality, or other patterns that can affect the performance of RNNs. The optimization and enhancement recommendations on MLPs are also valid on RNNs.

Long Short-Term Memory

Long short-term memory (LSTM) is a type of RNN that addresses the vanishing gradient problem and allows the network to capture long-term dependencies in sequential data. LSTMs were introduced by Hochreiter and Schmidhuber in 1997.

LSTMs are designed to overcome the limitations of traditional RNNs when dealing with long sequences of data. They achieve this by incorporating specialized memory cells that can retain information over extended time periods. The key idea behind LSTMs is the use of a gating mechanism that controls the flow of information through the memory cells.

The LSTM architecture consists of memory cells, input gates, forget gates, and output gates. The memory cells store and update information at each time step, while the gates regulate the flow of information. Here's how LSTMs work:

Input gate

The input gate determines which information from the current time step should be stored in the memory cell. It takes the current input and the previous hidden state as inputs, and then it applies a sigmoid activation function to generate a value between 0 and 1 for each component of the memory cell.

Forget gate

The forget gate determines which information from the previous memory cell should be forgotten. It takes the current input and the previous hidden state as inputs, and then it applies a sigmoid activation function to produce a forget vector. This vector is then multiplied element-wise with the previous memory cell values, allowing the LSTM to forget irrelevant information.

Update

The update step combines the information from the input gate and the forget gate. It takes the current input and the previous hidden state as inputs, and then it applies a tanh activation function. The resulting vector is then multiplied element-wise with the input gate output, and the product is added to the product of the forget gate and the previous memory cell values. This update operation determines which new information to store in the memory cell.

Output gate

The output gate determines the output of the LSTM at the current time step. It takes the current input and the previous hidden state as inputs, and then it applies a sigmoid activation function. The updated memory cell values are passed through a hyperbolic tangent (tanh) activation function and then multiplied element-wise with the output gate. The resulting vector becomes the current hidden state and is also the output of the LSTM at that time step.

The gating mechanisms in LSTMs allow them to selectively remember or forget information over long sequences, making them well suited for tasks involving long-term dependencies. By addressing the vanishing gradient problem and capturing long-term dependencies, LSTMs have become a popular choice for sequential data processing and have been instrumental in advancing the field of deep learning.



Theoretically, RNNs are capable of learning long-term dependencies, but in practice, they do not, hence the need for LSTMs.

As usual, let's apply LSTMs to the same time series problem. Note, however, that the results do not mean anything since the explanatory variables are arbitrary and the hyperparameters are not tuned. The aim of doing such exercises is to understand the code and the logic behind the algorithm. Afterward, it will be up to you to select the inputs and the variables that you deem worthy to be tested out.

Import the required libraries as follows:

```
from keras.models import Sequential
from keras.layers import Dense, LSTM
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Now set the hyperparameters of the model:

```
num_lags = 100
train_test_split = 0.80
num_neurons_in_hidden_layers = 20
num_epochs = 100
batch_size = 32
```

The LSTM model requires three-dimensional arrays of features. This can be done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

The following code block shows how to build the LSTM architecture in *keras*:

```
# Create the LSTM model
model = Sequential()
# First LSTM layer
model.add(LSTM(units = num_neurons_in_hidden_layers,
               input_shape = (num_lags, 1)))
# Second hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(units = 1))
# Compile the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Train the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
```

Let's plot the results and analyze the performance:

```
Accuracy Train = 65.63 %
Accuracy Test = 50.42 %
RMSE Train = 25.5619843783
RMSE Test = 55.1133475721
Correlation In-Sample Predicted/Train = 0.515
Correlation Out-of-Sample Predicted/Test = 0.057
Model Bias = 2.56
```

Figure 8-11 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`. Note that the hyperparameters are the same as the ones used in the RNN model.

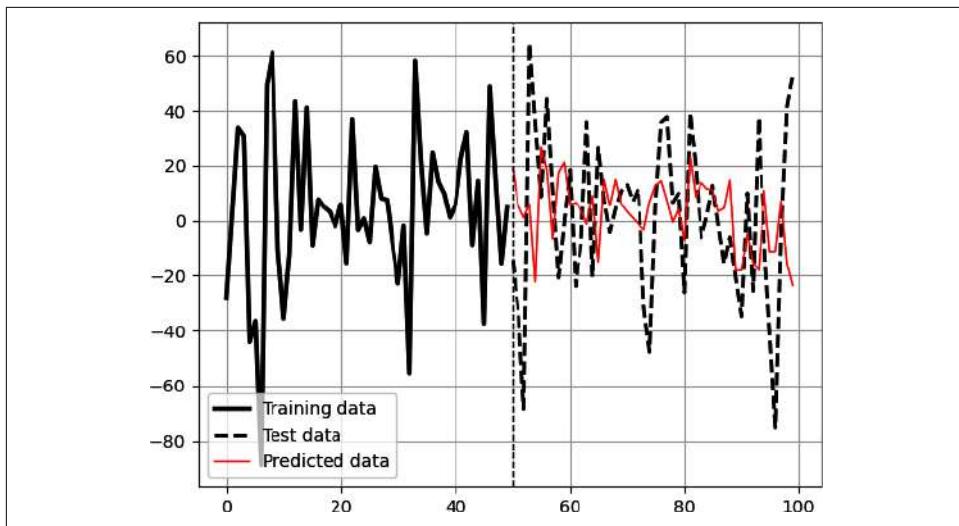


Figure 8-11. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the LSTM regression algorithm.

It is worth seeing how well the algorithm is fitted to the training data. [Figure 8-12](#) shows the values from `y_predicted_train` and `y_train`.

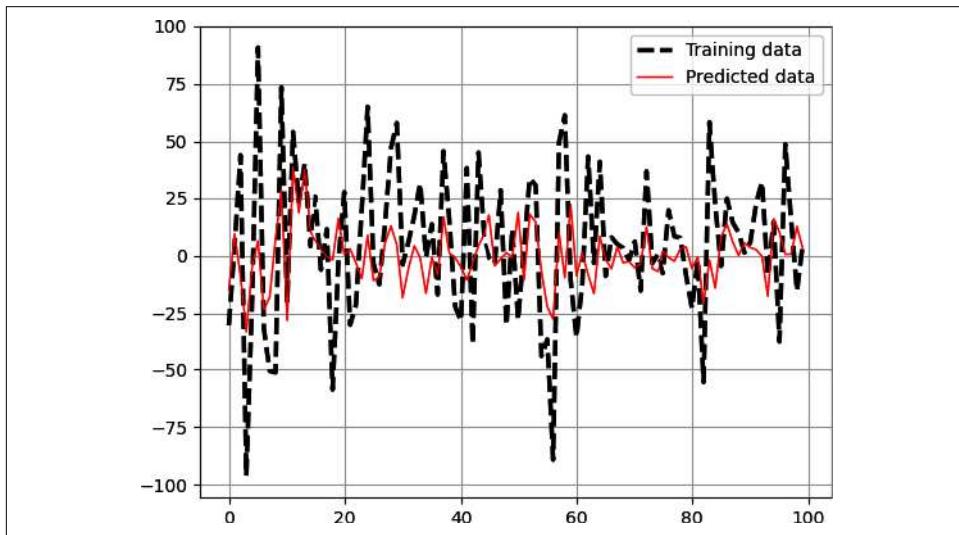


Figure 8-12. In-sample predictions using the LSTM regression algorithm.

In the context of LSTMs, a three-dimensional array represents the shape of the input data that is fed into the models. It is typically used to accommodate sequential or time series data in the form of input sequences. The dimensions of a three-dimensional array have specific meanings:

Dimension 1 (samples)

This dimension represents the number of samples or examples in the dataset. Each sample corresponds to a specific sequence or time series instance. For example, if you have 1,000 time series sequences in your dataset, dimension 1 would be 1,000.

Dimension 2 (time steps)

This dimension represents the number of time steps or data points in each sequence. It defines the length of the input sequence that the LSTM or RNN model processes at each time step. For instance, if your input sequences have a length of 10 time steps, dimension 2 would be 10.

Dimension 3 (features)

This dimension represents the number of features or variables associated with each time step in the sequence. It defines the dimensionality of each time step's data. In the case of univariate time series data, where only a single value is considered at each time step, dimension 3 would typically be 1. For multivariate time series, where multiple variables are observed at each time step, dimension 3 would be greater than 1.

Let's take a quick break and discuss an interesting topic. Using simple linear algorithms to model complex, nonlinear relationships is most likely to give bad results. At the same time, using extremely complex methods such as LSTMs on simple and predictable data may not be necessary even though it may provide positive results. [Figure 8-13](#) shows an ascending time series that looks like it's oscillating in regular intervals.

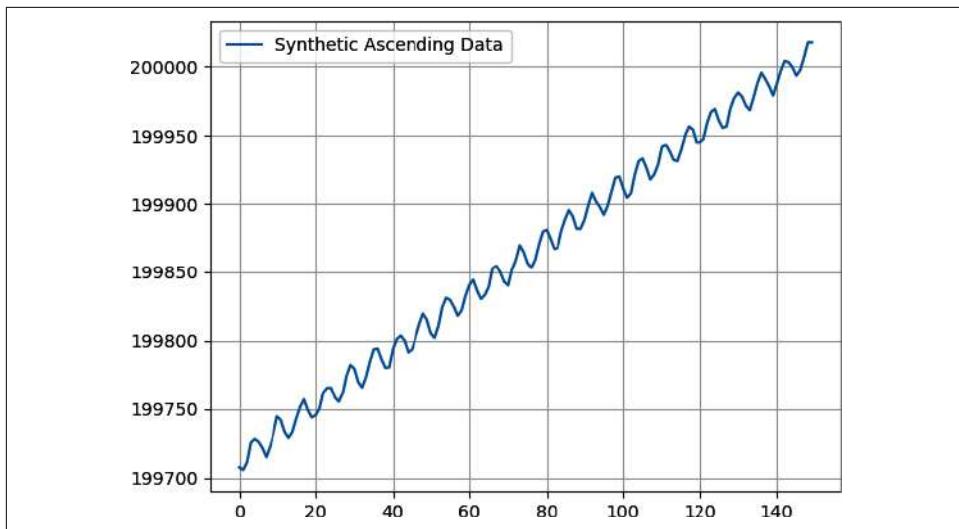


Figure 8-13. A generated ascending time series with oscillating properties.

Believe it or not, linear regression can actually model this raw time series quite well. By assuming an autoregressive model with 100 features (which means that to predict the next value, the model looks at the last 100 values), the linear regression algorithm can be trained on in-sample data and output the out-of-sample results shown in Figure 8-14.

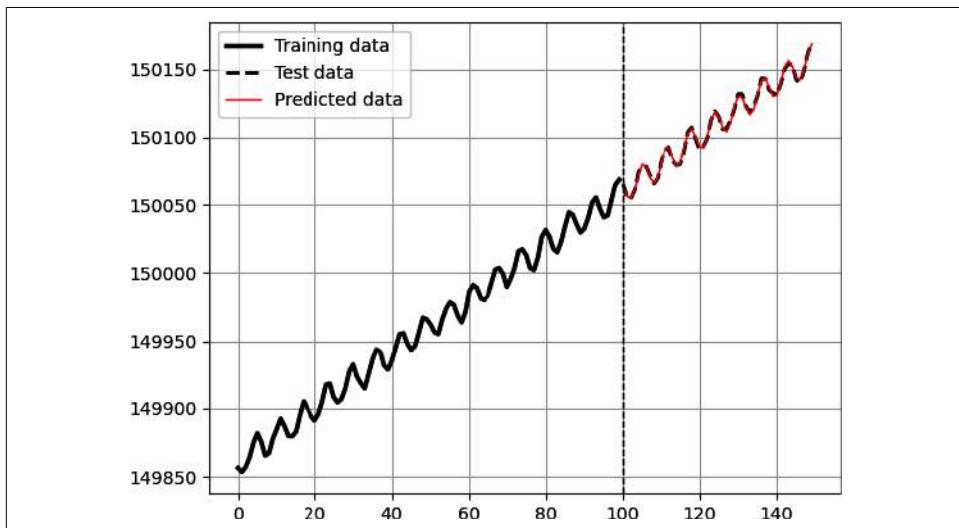


Figure 8-14. Prediction over the ascending time series using linear regression.

But let's take its first order difference and make it stationary. Take a look at [Figure 8-15](#), which shows a stationary time series created from differencing the time series shown in [Figure 8-13](#).

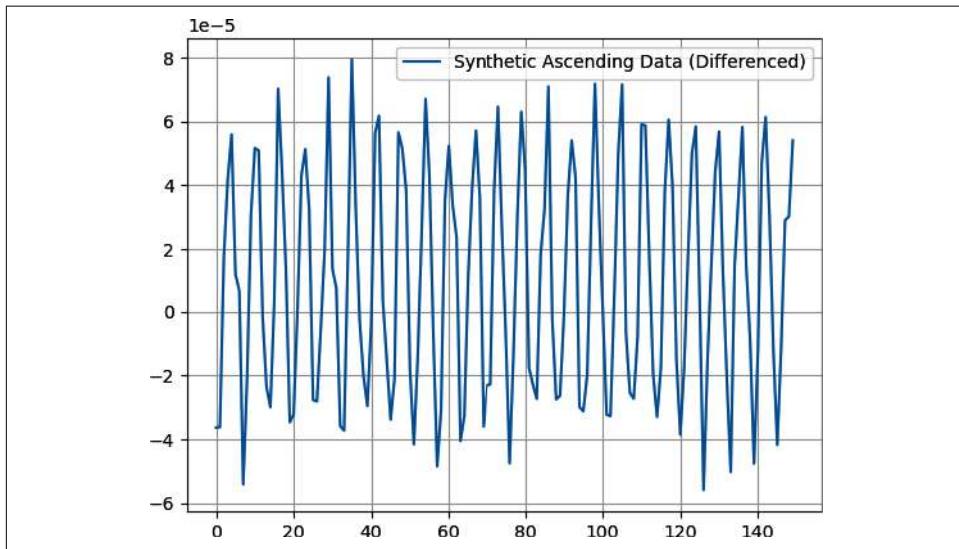


Figure 8-15. A generated ascending time series with oscillating properties (differenced).

The linear regression algorithm can be trained on in-sample data and output the out-of-sample results shown in [Figure 8-16](#) with extreme accuracy.

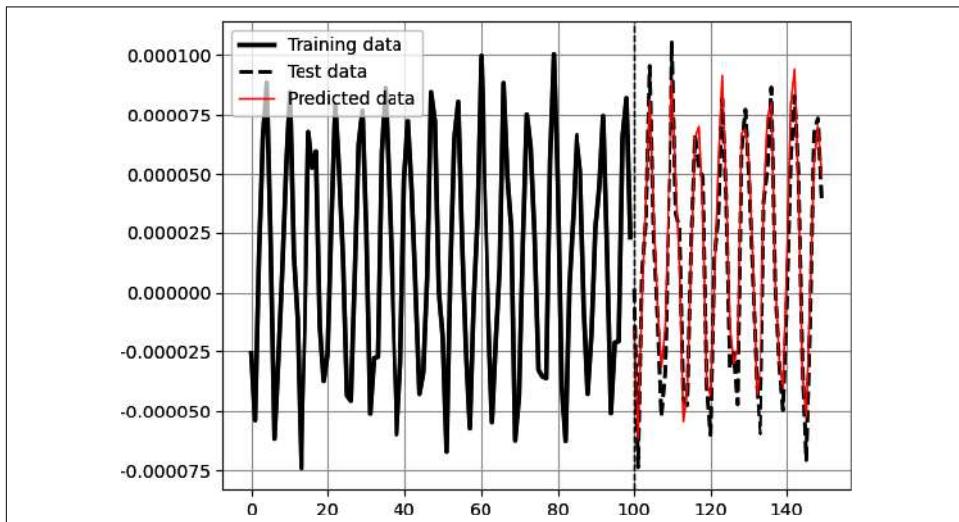


Figure 8-16. Prediction over the differenced time series using linear regression.

Another way of assessing the goodness of fit of a linear regression model is to use R^2 . Also known as the *coefficient of determination*, R^2 is a statistical measure that indicates the proportion of the variance in the dependent variable that can be explained by the independent variable(s) in a regression model.

R^2 ranges from 0 to 1 and is often expressed as a percentage. A value of 0 indicates that the independent variable(s) cannot explain any of the variability in the dependent variable, while a value of 1 indicates that the independent variable(s) can completely explain the variability in the dependent variable.

In simple terms, R^2 represents the proportion of the dependent variable's variability that can be attributed to the independent variable(s) included in the model. It provides a measure of how well the regression model fits the observed data. However, it does not indicate the causal relationship between variables or the overall quality of the model. It is also worth noting that R^2 is the squared correlation between the two variables. The R^2 metric for the differenced time series is 0.935, indicating extremely good fit.

In parallel, using an MLP with some optimization also yields good results. **Figure 8-17** shows the results of the differenced values when using a simple MLP model (with two hidden layers, each containing 24 neurons and a batch size of 128 run through 50 epochs).

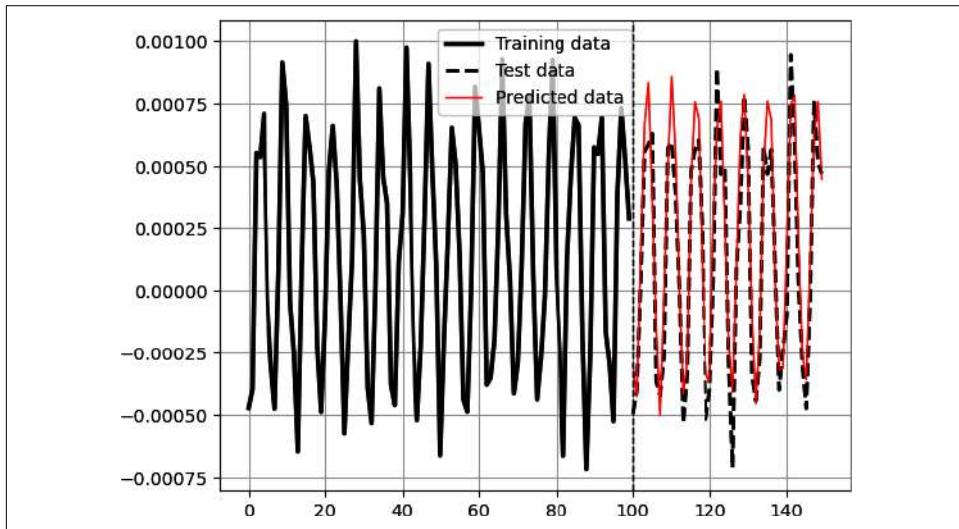


Figure 8-17. Prediction over the differenced time series using MLP.

However, the added complexity of using a deep learning method to predict such a simple time series may not be worth it.

Temporal Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep learning models designed to process structured grid-like data, with a particular emphasis on images and other grid-like data such as time series (less commonly used) and audio spectrograms. CNNs are good at learning and extracting hierarchical patterns and features from input data, making them powerful tools for tasks like image recognition, object detection, image segmentation, and more.

The core building blocks of CNNs are the *convolutional layers*. These layers perform convolution operations by applying a set of learnable filters to input data, resulting in feature maps that capture relevant spatial patterns and local dependencies. Another important concept with CNNs is *pooling layers*, which downsample the feature maps produced by convolutional layers. Common pooling operations include *max pooling* (selecting the maximum value in a neighborhood) and *average pooling* (computing the average value). Pooling helps reduce spatial dimensions, extract dominant features, and improve computational efficiency.



A CNN that is specifically used for time series forecasting is often referred to as a 1D-CNN or a *temporal convolutional network*.

The term *1D-CNN* indicates that the convolutional operations are applied along the temporal dimension of the input data, which is characteristic of time series data. This distinguishes it from traditional CNNs that operate on spatial dimensions in tasks such as image recognition.

A typical CNN architecture consists of three main components: an input layer, several alternating convolutional and pooling layers, and fully connected layers at the end. Convolutional layers are responsible for feature extraction, while pooling layers downsample the data. The fully connected layers provide the final predictions.

CNN architectures can vary greatly depending on the specific task. These architectures often employ additional techniques such as dropout regularization to improve performance and address challenges like overfitting.

CNNs can be used for time series prediction by leveraging their ability to capture local patterns and extract relevant features from the input data. The framework of the process is as follows:

1. CNNs use convolutional layers to perform localized feature extraction. The convolutional layers consist of a set of learnable filters that are convolved with the input data. Each filter extracts different features from the input data by applying element-wise multiplications and summations in a sliding window manner. The

result is a feature map that highlights important patterns or features at different locations in the input data.

2. Pooling layers are often employed after convolutional layers to reduce the spatial dimensionality of the feature maps. Max pooling is a common technique, where the maximum value within a local neighborhood is selected, effectively down-sampling the feature map. Pooling helps in capturing the most salient features while reducing the computational complexity and enhancing the network's ability to generalize.
3. After the convolutional and pooling layers, the resulting feature maps are typically flattened into a one-dimensional vector. This flattening operation transforms the spatially distributed features into a linear sequence, which can then be passed to fully connected layers.
4. Fully connected layers receive the flattened feature vector as input and learn to map it to the desired output. These layers enable the network to learn complex combinations of features and model the nonlinear relationships between input features and target predictions. The last fully connected layer typically represents the output layer, which predicts the target values for the time series.

Before moving to the algorithm creation steps, let's review some key concepts seen with CNNs. In time series forecasting with CNNs, *filters* are applied along the temporal dimension of the input data. Instead of considering spatial features as in image data, the filters are designed to capture temporal patterns or dependencies within the time series. Each filter slides across the time series, processing a subset of consecutive time steps at a time. The filter learns to detect specific temporal patterns or features in the input data. For example, it might capture short-term trends, seasonality, or recurring patterns that are relevant for the forecasting task. Multiple filters can be used in each convolutional layer, allowing the network to learn a diverse set of temporal features. Each filter captures different aspects of the time series, enabling the model to capture complex temporal relationships.

Another concept is the *kernel size*, which refers to the length or the number of consecutive time steps that the filter considers during the convolution operation. It defines the receptive field of the filter and influences the size of the extracted temporal patterns. The choice of kernel size depends on the characteristics of the time series data and the patterns to be captured. Smaller kernel sizes, such as 3 or 5, focus on capturing short-term patterns, while larger kernel sizes, such as 7 or 10, are suitable for capturing longer-term dependencies. Experimentation with different kernel sizes can help identify the optimal receptive field that captures the relevant temporal patterns for accurate forecasting. It's common to have multiple convolutional layers with different kernel sizes to capture patterns at various temporal scales.

Now let's see how to create a temporal CNN to forecast S&P 500 returns using its lagged values. Import the required libraries as follows:

```

from keras.models import Sequential
from keras.layers import Conv1D, MaxPooling1D, Flatten, Dense
import keras
import numpy as np
import matplotlib.pyplot as plt
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error

```

Next, set the hyperparameters of the model:

```

num_lags = 100
train_test_split = 0.80
filters = 64
kernel_size = 4
pool_size = 2
num_epochs = 100
batch_size = 8

```

Reshape the features arrays into three-dimensional data structures:

```

x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))

```

Now create the architecture of the temporal convolutional network (TCN) and run the algorithm:

```

# Create the temporal convolutional network model
model = Sequential()
model.add(Conv1D(filters = filters, kernel_size = kernel_size,
                 activation = 'relu', input_shape = (num_lags, 1)))
model.add(MaxPooling1D(pool_size = pool_size))
model.add(Flatten())
model.add(Dense(units = 1))
# Compile the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Train the model
model.fit(x_train, y_train, epochs = num_epochs , batch_size = batch_size)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))

```

Let's plot the results and analyze the performance:

```

Accuracy Train = 68.9 %
Accuracy Test = 49.16 %
RMSE Train = 18.3047790152
RMSE Test = 63.4069105299
Correlation In-Sample Predicted/Train = 0.786
Correlation Out-of-Sample Predicted/Test = 0.041
Model Bias = 0.98

```

Figure 8-18 shows the evolution of the forecasting task from the last values of `y_train` to the first values of `y_test` and `y_predicted`.

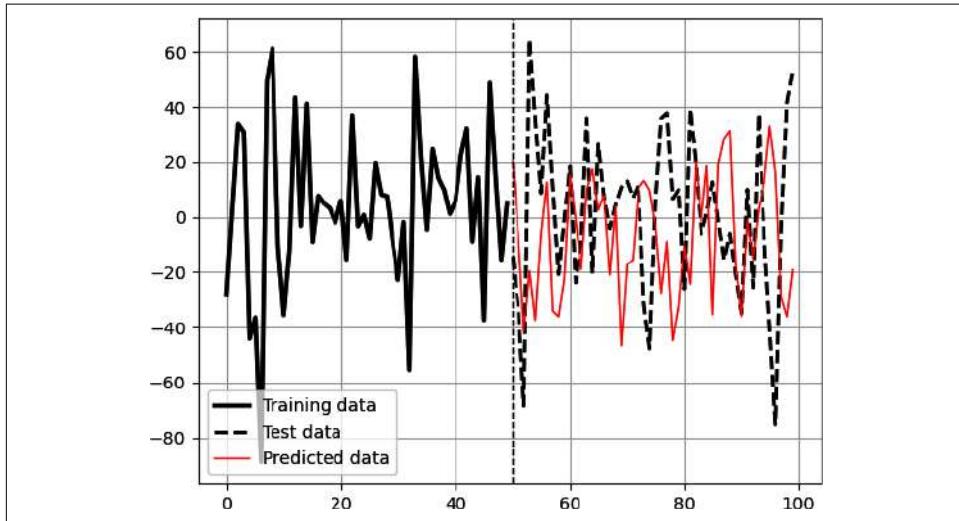


Figure 8-18. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the CNN regression algorithm.

It is important to use performance metrics that reflect your choice and to search for a better algorithm. Accuracy may be one of the base metrics to give you a quick glance at the predictive abilities of your model, but on its own, it is not enough. The results seen in this chapter reflect only the training using the selected hyperparameters. Optimization will allow you to achieve very good results on certain models.



There is no strict rule defining the number of hidden layers required to consider a neural network as deep. However, a common convention is that a neural network with two or more hidden layers is typically considered a deep neural network.

Summary

Applying deep learning algorithms to time series data can offer several benefits and challenges. Deep learning algorithms have shown great utility in time series analysis by effectively capturing complex patterns, extracting meaningful features, and making accurate predictions. However, their success relies heavily on the quality of the data and the chosen features.

The utility of applying deep learning algorithms on time series data stems from their ability to automatically learn hierarchical representations and model intricate temporal dependencies. They can handle nonlinear relationships and capture both local and global patterns, making them suitable for a wide range of time series tasks like forecasting, anomaly detection, classification, and signal processing.

However, applying deep learning algorithms to time series can present challenges:

Data quality

Deep learning models heavily rely on large amounts of high-quality, labeled data for training. Insufficient or noisy data can hinder the performance of the models, leading to inaccurate predictions or unreliable insights. Data preprocessing, cleaning, and addressing missing values become crucial steps to ensure the quality of the data.

Feature engineering

Deep learning models can automatically learn relevant features from the data. However, the choice and extraction of informative features can significantly impact the model's performance. Domain knowledge, data exploration, and feature engineering techniques are important in selecting or transforming features that enhance the model's ability to capture relevant patterns.

Model complexity

Deep learning models are typically complex with a large number of parameters. Training such models requires substantial computational resources, longer training times, and careful hyperparameter tuning. Overfitting, where the model memorizes the training data without generalizing well to unseen data, is also a common challenge.

Interpretability

Deep learning models are often considered mystery boxes, making it challenging to interpret the learned representations and understand the reasoning behind predictions. This can be a concern in domains where interpretability and explainability are crucial, such as finance.

To overcome these challenges and harness the power of deep learning algorithms for time series analysis, careful consideration of data quality, appropriate feature engineering, model architecture selection, regularization techniques, and interpretability approaches are essential. It is crucial to understand the specific characteristics and requirements of the time series data and the task at hand to choose and tailor the deep learning approach accordingly.

Deep Learning for Time Series Prediction II

This chapter presents a few techniques and methods to complement the forecasting task of machine and deep learning algorithms. It is composed of different topics that each discuss a way to improve and optimize the process. At this point, you should have a sound understanding of the basics of machine and deep learning models, and you know how to code a basic algorithm that predicts the returns of a financial time series (or any stationary time series). This chapter bridges the gap between the basic knowledge and the advanced knowledge required to elevate the algorithms to a functional level.

Fractional Differentiation

In his book *Advances in Financial Machine Learning*, Marcos López de Prado describes a technique to transform nonstationary data into stationary data. This is referred to as fractional differentiation.

Fractional differentiation is a mathematical technique used to transform a time series into a stationary series while preserving some of its memory. It extends the concept of *differencing* (or taking the returns), which is commonly used to remove trends and make time series stationary.

In traditional differencing, the data sequence is differenced by a whole number, typically 1, which involves subtracting the previous value from the current value. This helps eliminate trends and makes the series stationary. However, in some cases, the series may exhibit long-term dependencies or memory effects that are not effectively captured by traditional differencing. These dependencies may help in forecasting the time series, and if they are completely eliminated, that may hinder the ability of the algorithm to perform well. These dependencies are referred to as *memory*.

Fractional differentiation addresses this limitation by allowing the differencing parameter to be a fractional value. The fractional differencing operator effectively applies a weighted sum of lagged values to each observation in the series, with the weights determined by the fractional differencing parameter. This allows for capturing long-term dependencies or memory effects in the series. Fractional differentiation is particularly useful in financial time series analysis, where data often exhibits long memory or persistent behavior. This can be implemented in Python. First, pip install the required library from the prompt:

```
pip install fracdiff
```

Next, import the required libraries:

```
from fracdiff.sklearn import Fracdiff
import pandas_datareader as pdr
import numpy as np
import matplotlib.pyplot as plt
```

Let's use the classic example that de Prado uses in his book, the S&P 500, to prove that fractional differentiation transforms a nonstationary time series into a stationary one with visible preserved memory.

The following code applies fractional differentiation and compares it to traditional differencing:

```
# Set the start and end dates for the data
start_date = '1990-01-01'
end_date   = '2023-06-01'
# Fetch S&P 500 price data
data = np.array((pdr.get_data_fred('SP500', start = start_date,
                                    end = end_date)).dropna())
# Calculate the fractional differentiation
window = 100
f = Fracdiff(0.48, mode = 'valid', window = window)
frac_data = f.fit_transform(data)
# Calculate a simple differencing function for comparison
diff_data = np.reshape(np.diff(data[:, 0]), (-1, 1))
# Harmonizing time indices
data = data[window - 1:, :]
diff_data = diff_data[window - 2:, :]
```

Figure 9-1 shows the three types of transformations. You can notice the trending nature in the top panel with the nontransformed S&P 500 data. You can also notice that in the middle panel, this trend is less visible but still there. This is what fractional differentiation aims to do. By keeping a hint of the market's memory while rendering it stationary, this technique can help improve some forecasting algorithms. The bottom panel shows normal differencing of the price data.

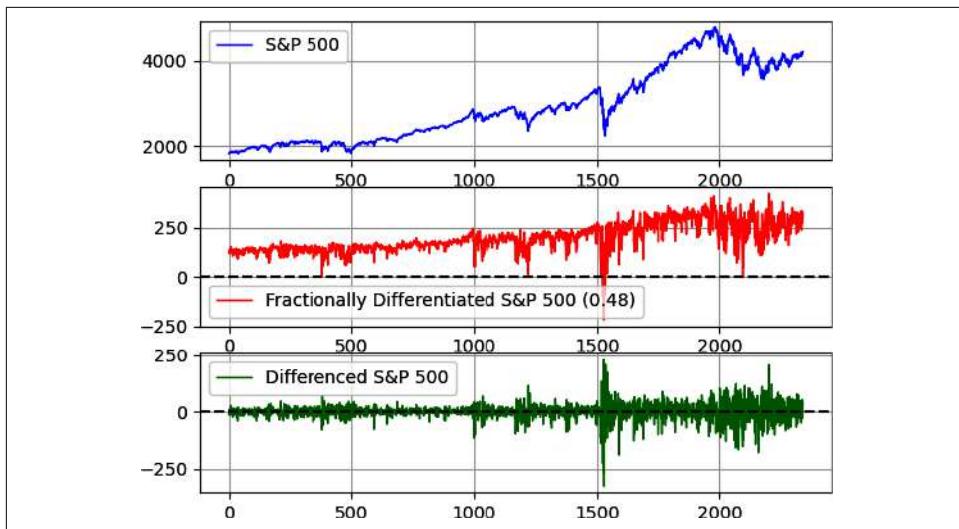


Figure 9-1. Fractional differentiation on S&P 500 (order = 0.48)

Figure 9-1 was generated using this code:

```
fig, axes = plt.subplots(nrows = 3, ncols = 1)
axes[0].plot(data[5:,], label = 'S&P 500', color = 'blue', linewidth = 1)
axes[1].plot(frac_data[5:,], label =
              'Fractionally Differentiated S&P 500 (0.48)',
              color = 'orange', linewidth = 1)
axes[2].plot(diff_data[5:,], label =
              'Differenced S&P 500', color = 'green', linewidth = 1)
axes[0].legend()
axes[1].legend()
axes[2].legend()
axes[0].grid()
axes[1].grid()
axes[2].grid()
axes[1].axhline(y = 0, color = 'black', linestyle = 'dashed')
axes[2].axhline(y = 0, color = 'black', linestyle = 'dashed')
```

Let's make sure that the fractionally differentiated data is indeed stationary by applying the augmented Dickey–Fuller (ADF) test (you used this test in [Chapter 3](#)):

```
from statsmodels.tsa.stattools import adfuller
print('p-value: %f' % adfuller(data)[1])
print('p-value: %f' % adfuller(frac_data)[1])
print('p-value: %f' % adfuller(diff_data)[1])
```

The output of the previous code block is as follows (assuming a 5% significance level):

```
# The original S&P 500 dataset is nonstationary
p-value: 0.842099
# The fractionally differentiated S&P 500 dataset is stationary
p-value: 0.038829
# The normally differenced S&P 500 dataset is stationary
p-value: 0.000000
```

As the results show, the data is indeed stationary. Let's look at another example. The following code imports the daily values of the EURUSD:

```
data = np.array((pdr.get_data_fred('DEXUSEU', start = start_date,
                                    end = end_date)).dropna())
```

Figure 9-2 compares the EURUSD with fractional differentiation (0.20) applied onto it, with the regular differencing shown in the bottom panel.

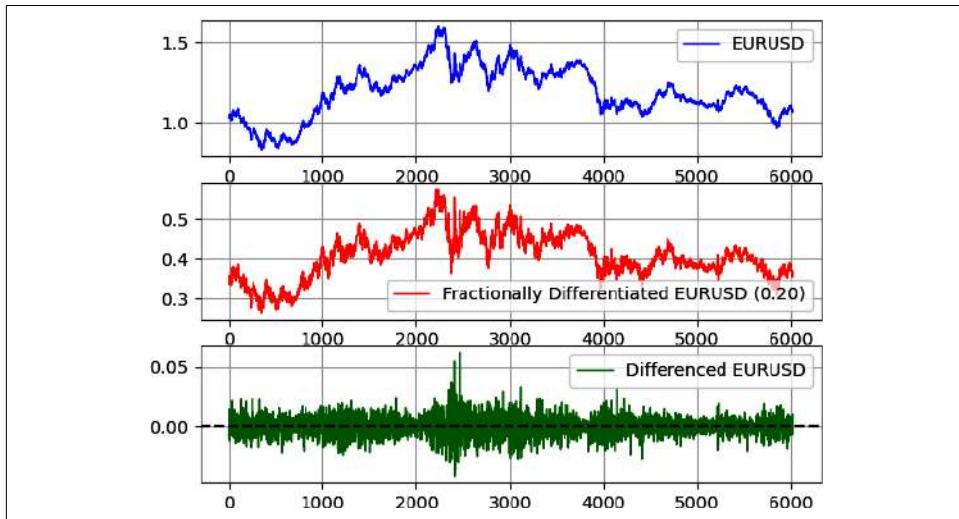


Figure 9-2. Fractional differentiation on the EURUSD (order = 0.20)

The results of the ADF test are as follows:

```
# The original EURUSD dataset is nonstationary
p-value: 0.397494
# The fractionally differentiated EURUSD dataset is stationary
p-value: 0.043214
# The normally differenced EURUSD dataset is stationary
p-value: 0.000000
```

As a comparison, [Figure 9-3](#) compares the same dataset with fractional differentiation (0.30) applied onto it, with the regular differencing shown in the bottom panel.

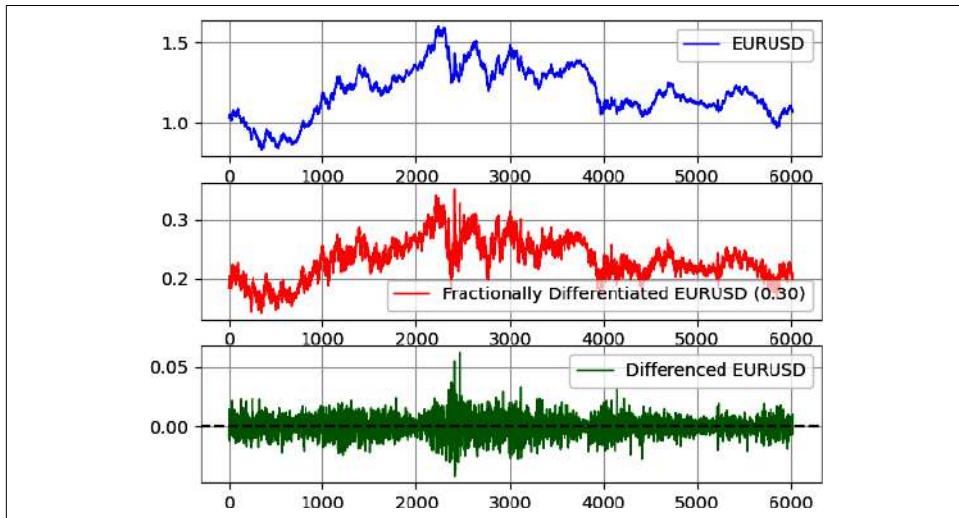


Figure 9-3. Fractional differentiation on EURUSD (order = 0.30)



Approaching an order of 1.00 intuitively makes the fractional differentiation approach a normal integer differencing. Similarly, approaching an order of 0.00 makes the fractional differentiation approach the untransformed data series.

[Figure 9-3](#) shows a more stationary EURUSD series in the middle panel than [Figure 9-2](#) does, and this is because the order of fractional differentiation is increased. This is why the ADF test result for the fractional differentiation of order = 0.30 is 0.002, which is much lower than the ADF test result when order = 0.20 (which is at 0.043).

In summary, fractional differentiation is a valuable tool for time series prediction as it captures long-term dependencies, handles nonstationarity, adapts to various dynamics, and preserves integral properties. Its ability to capture complex patterns and improve forecasting accuracy makes it a good fit for modeling and predicting a wide range of real-world time series data.

Forecasting Threshold

The *forecasting threshold* is the minimum required percentage prediction to validate a signal. This means that the forecasting threshold technique is a filter that removes low conviction predictions.

Objectively, low conviction predictions are below a certain percentage. A hypothetical example is shown in [Table 9-1](#). The threshold is $\pm 1\%$.

Table 9-1. Table of forecasts

Time	Forecast	Status
1	0.09%	Dismissed
2	-0.60%	Dismissed
3	-1.50%	Taken
4	1.00%	Taken
5	2.33%	Taken

At time 1, the trading signal is bullish, with an expectation of a 0.09% rise in the hypothetical financial instrument. As this prediction is below the threshold of 1.00%, the trade is not taken. At time 2, the same intuition is applied, as the bearish signal is below the threshold.

The rest of the signals are taken since they are equal to or greater than the threshold (in terms of magnitude). The aim of this section is to develop a multilayer perceptron (MLP) model and keep only the predictions that respect a certain threshold.

As usual, start by importing the required libraries:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from master_function import data_preprocessing, mass_import
from master_function import plot_train_test_values, forecasting_threshold
```

Next, set the hyperparameters and import the data using `mass_import()`:

```
num_lags = 500
train_test_split = 0.80
num_neurons_in_hidden_layers = 256
num_epochs = 100
batch_size = 10
threshold = 0.0015
```

Import and preprocess the data, then design the MLP architecture:

```
# Fetching the historical price data
data = np.diff(mass_import(0, 'D1')[ :, 3])
# Creating the training and test sets
```

```

x_train, y_train, x_test, y_test = data_preprocessing(data, num_lags,
                                                    train_test_split)

# Designing the architecture of the model
model = Sequential()
# First hidden layer
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
                activation = 'relu'))
# Second hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(1))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')

```

The next step is to fit and predict the data and retain the predictions that satisfy the threshold you have defined in the hyperparameters. This is done using the function `forecasting_threshold()`:

```

# Fitting
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting
y_predicted = model.predict(x_test)
# Threshold function
y_predicted = forecasting_threshold(y_predicted, threshold)
# Plotting
plot_train_test_values(100, 50, y_train, y_test, y_predicted)

```

Figure 9-4 shows the comparison chart between the real values and the predicted values. Flat observations on the predictions indicate the absence of signals that are lower than the required threshold—in this case, 0.0015.

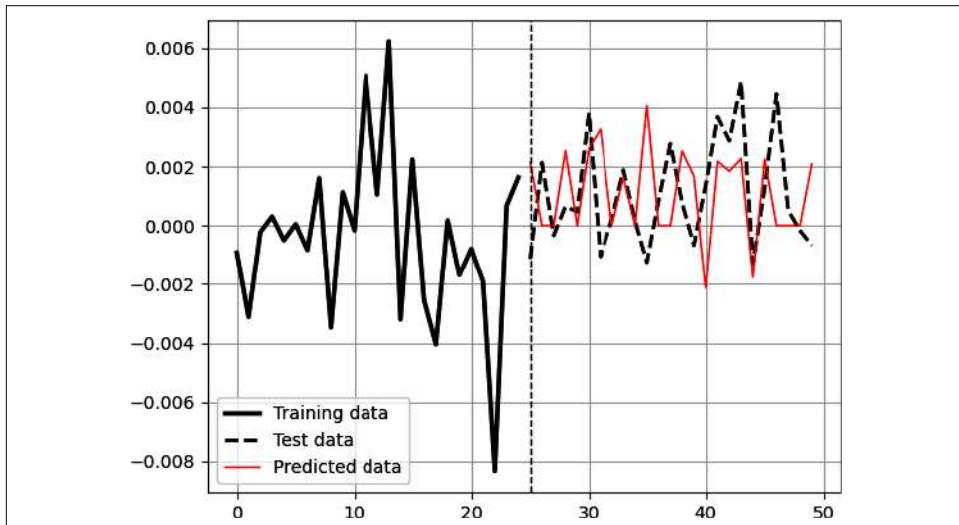


Figure 9-4. Predicting with the forecasting threshold

The threshold can be found in many ways, notably:

The fixed numerical technique

As you saw in the previous example, this technique assumes a fixed arbitrary number to be used as a threshold.

The volatility-based technique

With this technique, you use a volatility indicator such as a rolling standard deviation of prices to set a variable threshold at each time step. This technique has the benefit of using up-to-date volatility information.

The statistical technique

With this technique, you look at the real values from the training set (not the test set) and select a certain quantile (e.g., the 75% quantile) as a minimum threshold to validate the signals.

To summarize, using the forecasting threshold may help select the trades with the highest conviction and can also help minimize transaction costs since the algorithms assume trading all the time, which is not recommended. This assumes adding a new state to the algorithm, which gives a total of three:

Bullish signal

The algorithm predicts a higher value.

Bearish signal

The algorithm predicts a lower value.

Neutral signal

The algorithm does not have any directional conviction.

Continuous Retraining

Retraining refers to the act of training the algorithm every time new data comes in. This means that when dealing with a daily time series, the retraining is done every day while incorporating the latest daily inputs.

The continuous retraining technique deserves to be tested, and that is the aim of this section. The architecture of the algorithm will follow this framework:

1. Train the data on the training test.
2. For each prediction made, rerun the algorithm and include the new real inputs in the training set.



One big limitation of the continuous retraining technique is the speed of the algorithm, as it has to retrain at every time step. If you have 1,000 instances of test data where every training requires a few minutes, then the backtesting process becomes drastically slow. This is especially an issue with deep learning algorithms such as LSTM, which may take a long time to train.

The main reason for applying continuous retraining is because of *concept drift*, which is the change in the data's inner dynamics and structures that may invalidate the function found in the training phase. Basically, financial time series do not exhibit static relationships; rather, they change over time. Therefore, continuous retraining aims to update the models by always using the latest data to train.



Continuous retraining does not need to be done at every time step. You can set n periods for the retraining. For example, if you select 10, then the model retrains after each group of 10 new values.

To simplify things, this section shows the code for the continuous retraining (every day) using a linear regression model on the weekly EURUSD values at every time step. You can do the same thing with other models; you just have to change the lines of code where the model is imported and designed. First, import the required libraries:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression
from master_function import data_preprocessing, mass_import
from master_function import plot_train_test_values,
from master_function import calculate_accuracy, model_bias
from sklearn.metrics import mean_squared_error
```

Import the data and set the hyperparameters of the algorithm:

```
# Importing the time series
data = np.diff(mass_import(0, 'D1')[:, 3])
# Setting the hyperparameters
num_lags = 15
train_test_split = 0.80
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data, num_lags,
                                                      train_test_split)
# Fitting the model
model = LinearRegression()
model.fit(x_train, y_train)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
```

Create the continuous retraining loop as follows:

```
# Store the new forecasts
y_predicted = []
# Reshape x_test to forecast one period
latest_values = np.transpose(np.reshape(x_test[0], (-1, 1)))
# Isolate the real values for comparison
y_test_store = y_test
y_train_store = y_train
for i in range(len(y_test)):
    try:
        # Predict over the first x_test data
        predicted_value = model.predict(latest_values)
        # Store the prediction in an array
        y_predicted = np.append(y_predicted, predicted_value)
        # Add the first test values to the last training values
        x_train = np.concatenate((x_train, latest_values), axis = 0)
        y_train = np.append(y_train, y_test[0])
        # Remove the first test values from the test arrays
        y_test = y_test[1:]
        x_test = x_test[1:, ]
        # Retrain
        model.fit(x_train, y_train)
        # Select the first values of the test set
        latest_values = np.transpose(np.reshape(x_test[0], (-1, 1)))
    except IndexError:
        pass
```

Plot the predicted values:

```
plot_train_test_values(100, 50, y_train, y_test_store, y_predicted)
```

Figure 9-5 shows the result.

As a simple comparison, the same backtest was done on the model with no retraining. The latter got a 48.55% test set accuracy compared to the 48.92% test set accuracy for the same model with retraining.

Continuous retraining is not a guarantee for better results, but it makes sense to update the model every once in a while due to changing market dynamics. The frequency at which you should update the model may be subjective.

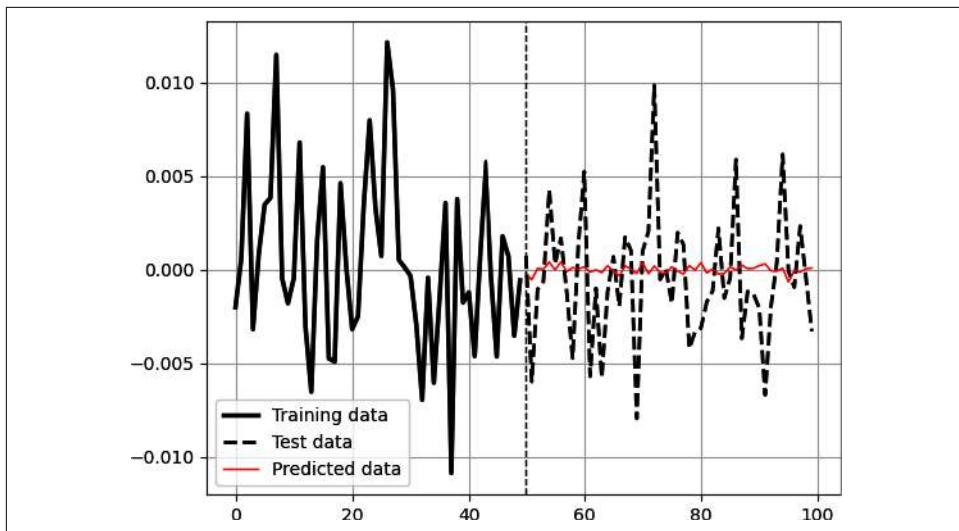


Figure 9-5. Predicting using the continuous retraining technique

Time Series Cross Validation

Cross validation is a technique used in machine learning to assess the performance of a model. It involves splitting the available data into subsets for training and evaluation. In the case of time series data, where the order of observations is important (due to the sequential nature of the data), a traditional k -fold cross validation approach may not be suitable. Instead, time series cross validation techniques are used, such as the *rolling window* and *expanding window* methods.



In traditional *k -fold cross validation*, the data is randomly split into k equally sized folds. Each fold is used as a validation set, while the remaining $k - 1$ folds are combined for training the model. The process is repeated k times, with each fold serving as the validation set once. Finally, the performance metrics are averaged across the k iterations to assess the model's performance.

Unlike traditional k -fold cross validation, time series cross validation methods respect the temporal order of data points. Two commonly used techniques for time series cross validation are the rolling window and expanding window methods.

In *rolling window cross validation*, a fixed-size training window is moved iteratively over the time series data. At each step, the model is trained on the observations within the window and evaluated on the subsequent window. This process is repeated until the end of the data is reached. The window size can be defined based on a

specific time duration or a fixed number of observations. [Figure 9-6](#) shows an illustration of rolling window cross validation.

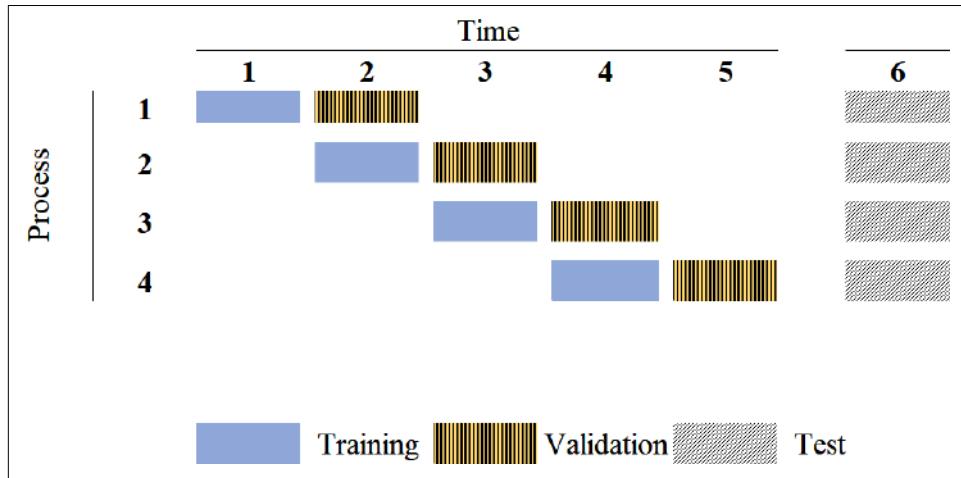


Figure 9-6. Rolling window cross validation

In *expanding window cross validation*, the training set starts with a small initial window and expands over time, incorporating additional data points at each step. The model is trained on the available data up to a specific point and evaluated on the subsequent time period. Similar to the rolling window approach, this process is repeated until the end of the data is reached. [Figure 9-7](#) shows an illustration of expanding window cross validation.

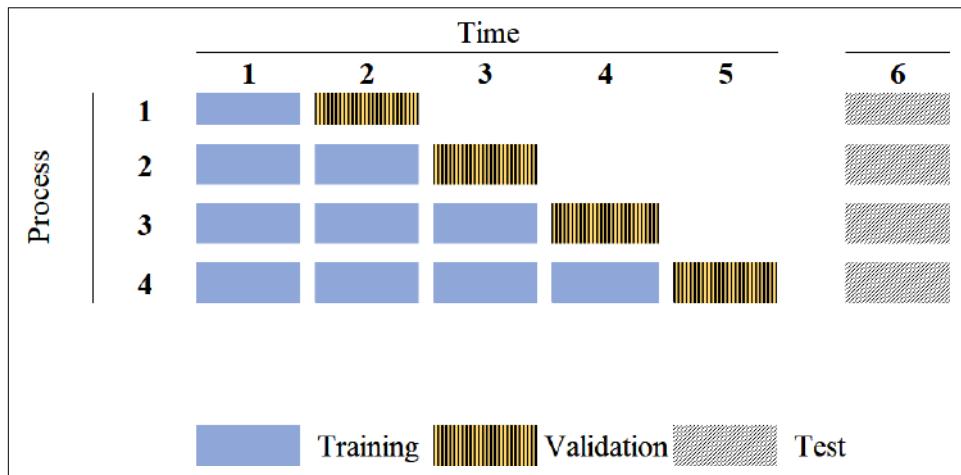


Figure 9-7. Expanding window cross validation

During each iteration of time series cross validation, the model's performance is measured using appropriate evaluation metrics. The performance results obtained from each iteration can be aggregated and summarized to assess the model's overall performance on the time series data.

Multiperiod Forecasting

Multiperiod forecasting (MPF) is a technique that aims to forecast more than just the next period. It aims to generate a path with n periods as defined by the user. There are two ways to approach MPF:

Recursive model

The *recursive model* uses the prediction as an input for the next prediction. As you may have already guessed, the recursive model may quickly get off track due to the exponentially rising error term from predicting while using predictions as inputs.

Direct model

The *direct model* trains the model from the beginning into outputting multiple forecasts in their respective time periods. This model is likely to be more robust than the recursive model.

Let's start with the recursive model. Mathematically speaking, its most basic form can be represented as follows:

$$\text{Prediction}_i = f(\text{Prediction}_{i-1}, \dots, \text{Prediction}_{i-n})$$

This section will use weather data and an economic indicator to apply the deep learning algorithm.

The first step in predictive analysis is to get to know the data, so let's see what the algorithm will aim to forecast. The first time series is the average daily temperature in Basel, Switzerland, since 2005. [Figure 9-8](#) shows the time series.

The second time series is the Institute for Supply Management's Purchasing Managers' Index (ISM PMI), a widely recognized economic indicator in the United States that provides insight into the health of the manufacturing sector and the overall economy. The index is based on a monthly survey of purchasing managers from various industries, including manufacturing, and assesses key factors such as new orders, production, employment, supplier deliveries, and inventories.

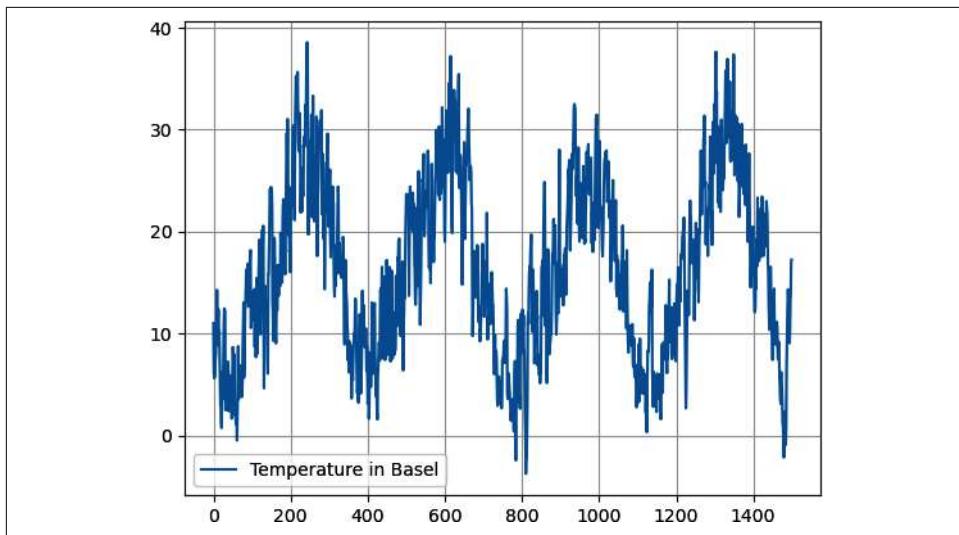


Figure 9-8. A sample from the dataset showing the seasonal nature of temperature

The index is reported as a percentage, with a value above 50 indicating expansion in the manufacturing sector and a value below 50 suggesting contraction. A higher PMI typically indicates positive economic growth, while a lower PMI may signal economic slowdown or recessionary conditions. The ISM PMI is closely monitored by policy-makers, investors, and businesses as it can offer valuable insights into economic trends and potential shifts in the business cycle. [Figure 9-9](#) shows the ISM PMI historical observations.

The aim of the forecast is to test the algorithm's ability to push through the noise and model the original mean-reverting nature of the ISM PMI. Let's start with the recursive model.

The framework for the recursive model is as follows:

1. Train the data on the training set using the usual 80/20 split.
2. Forecast the first observation using the inputs needed from the test set.
3. Forecast the second observation using the last prediction in step 2 and the required data from the test set while dropping the first observation.
4. Repeat step 3 until reaching the desired number of predictions. At some point, a prediction is made by solely looking at previous predictions.

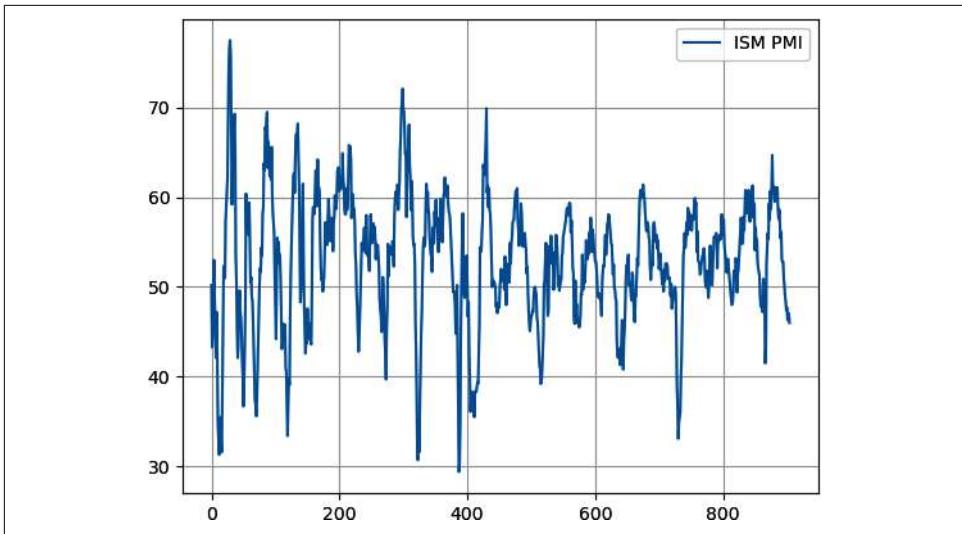


Figure 9-9. A sample from the imported dataset showing the mean-reverting nature of the ISM PMI



Up until now, you have been evaluating accuracy using `calculate_accuracy()`, which works when you are predicting positive or negative values (such as EURUSD price changes). When dealing with multiperiod forecasting of values that do not hover around zero, it is better to calculate the directional accuracy, which is basically the same calculation but does not hover around zero. For this, the function `calculate_directional_accuracy()` is used. Remember that the functions can be found in `master_function.py` in the book's [GitHub repository](#).

Let's start with the average temperature in Basel. Import the dataset using the following code (make sure you download the historical observations data from the [GitHub repository](#)):

```
from keras.models import Sequential
from keras.layers import Dense
import keras
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from master_function import data_preprocessing, plot_train_test_values,
from master_function import recursive_mpf
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
```

Next, preprocess the data:

```
# Importing the data
data = np.reshape(np.array(pd.read_excel('Temperature_Basel.xlsx')).dropna(),
                  (-1))
# Setting the hyperparameters
num_lags = 500
train_test_split = 0.8
num_neurons_in_hidden_layers = 100
num_epochs = 200
batch_size = 12
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data, num_lags,
                                                       train_test_split)
```

Design the architecture of the MLP with multiple hidden layers. Then, fit and predict on a recursive basis:

```
# Designing the architecture of the model
model = Sequential()
# First hidden layer
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
                activation = 'relu'))
# Second hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Third hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Fourth hidden layer
model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(1))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, np.reshape(y_train, (-1, 1)), epochs = num_epochs,
           batch_size = batch_size)
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting in the test set on a recursive basis
x_test, y_predicted = recursive_mpf(x_test, y_test, num_lags,
                                      model, architecture = 'MLP')
```

The `recursive_mpf()` function takes the following arguments:

- The test set features that will continuously be updated. They are represented by the variable `x_test`.
- The test set dependent variables. They are represented by the variable `y_test`.
- The number of lags. This variable is represented by `num_lags`.
- The fitted model as defined by the variable `model`.

- The type of architecture as represented by the argument `architecture`. It can either be `MLP` for two-dimensional arrays or `LSTM` for three-dimensional arrays.

Figure 9-10 shows the predictions versus the real values (the dashed time series after the cutoff line). Notice how the deep neural network re-creates the seasonal characteristics of the time series (albeit with some imperfections) and projects it well into the future with no required knowledge along the way.

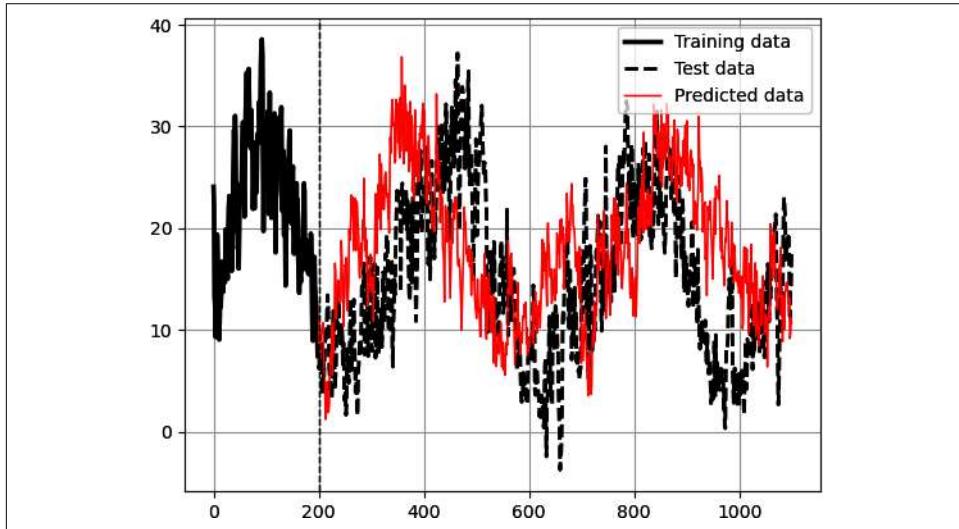


Figure 9-10. Multiperiod forecasts versus real values



Many machine and deep learning algorithms are able to model this relationship well. This example used MLPs, but this does not undermine other models, even simple ones such as linear regression. A good task for you would be to try applying the same example using a model of your choice (such as LSTM) and comparing the results. If you are using an LSTM model, make sure you set `architecture = 'LSTM'`.

Now apply the same process on the second time series. You only need to change the name of the imported file and the hyperparameters (as you see fit):

```
data = np.reshape(np.array(pd.read_excel('ISM_PMI.xlsx').dropna()), (-1))
```

Figure 9-11 shows the predictions (dashed line) versus the real values.

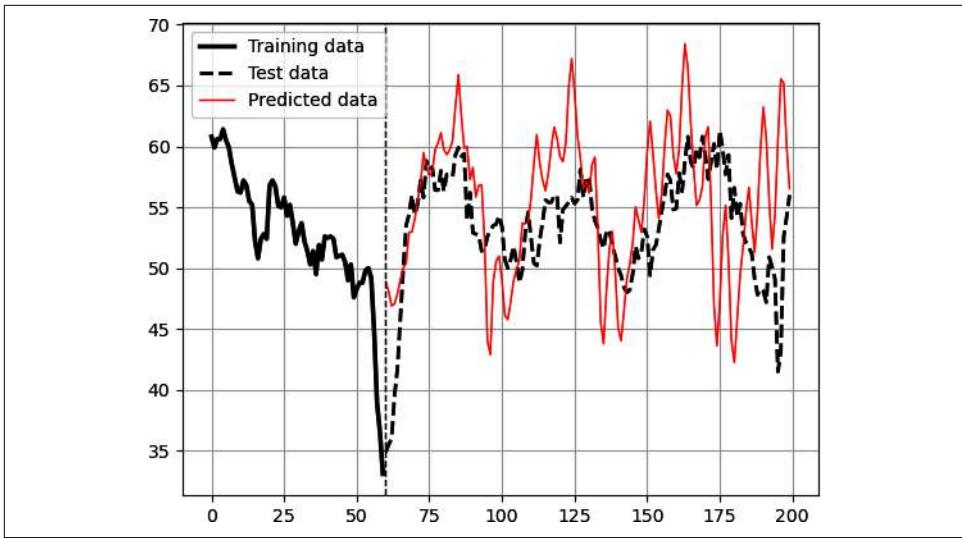


Figure 9-11. Forecasting multiple periods ahead; predicted data in thin solid line and test data in dashed line

The trained model is not too complex so as to avoid overfitting. However, it does manage to time turning points quite well during the first projections. Naturally, over time, this ability slowly fades away. Tweaking the hyperparameters is the key to achieving good directional accuracy. Start with the following hyperparameters:

```
num_lags = 200
train_test_split = 0.8
num_neurons_in_hidden_layers = 500
num_epochs = 400
batch_size = 100
```

The second MPF technique trains the model from the beginning into outputting multiple forecasts in their respective time periods. Mathematically, it can be represented as follows:

$$\text{Prediction}_i = fx(\text{real input}_{i-1}, \dots, \text{input}_{i-n})$$

$$\text{Prediction}_{i+1} = fx(\text{real input}_{i-1}, \dots, \text{input}_{i-n})$$

$$\text{Prediction}_{i+2} = fx(\text{real input}_{i-1}, \dots, \text{input}_{i-n})$$

The framework for the recursive model is as follows:

1. Create a function that relates the desired number of inputs to the desired number of outputs. This means that the last layer of the neural network will contain a

number of outputs equal to the number of forecasting periods you want to project into the future.

2. Train the model to predict multiple outputs at every time step based on the inputs from the same time step.

Let's continue with the ISM PMI. As usual, import the required libraries:

```
from keras.models import Sequential
from keras.layers import Dense
import keras
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from master_function import direct_mpf
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
```

Import and preprocess the data while setting the hyperparameters:

```
# Importing the data
data = np.reshape(np.array(pd.read_excel('ISM_PMI.xlsx').dropna()), (-1))
# Setting the hyperparameters
num_lags = 10
train_test_split = 0.80
num_neurons_in_hidden_layers = 200
num_epochs = 200
batch_size = 10
forecast_horizon = 18 # This means eighteen months
x_train, y_train, x_test, y_test = direct_mpf(data, num_lags,
                                               train_test_split,
                                               forecast_horizon)
```

The `direct_mpf()` function takes the following arguments:

- The dataset represented by the variable `data`
- The number of lags represented by the variable `num_lags`
- The split represented by the variable `train_test_split`
- The number of observations to project represented by the variable `forecast_horizon`

Prepare the arrays, design the architecture, and predict the data for a horizon of 18 months (since the ISM PMI is a monthly indicator):

```
# Designing the architecture of the model
model = Sequential()
# First hidden layer
model.add(Dense(num_neurons_in_hidden_layers, input_dim = num_lags,
               activation = 'relu'))
# Second hidden layer
```

```

model.add(Dense(num_neurons_in_hidden_layers, activation = 'relu'))
# Output layer
model.add(Dense(forecast_horizon))
# Compiling
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting (training) the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Make predictions
y_predicted = model.predict(x_test)
# Plotting
plt.plot(y_predicted[-1], label = 'Predicted data', color = 'red',
          linewidth = 1)
plt.plot(y_test[-1], label = 'Test data', color = 'black',
          linestyle = 'dashed', linewidth = 2)
plt.grid()
plt.legend()

```

Figure 9-12 shows the predicted data and the test data at this point.

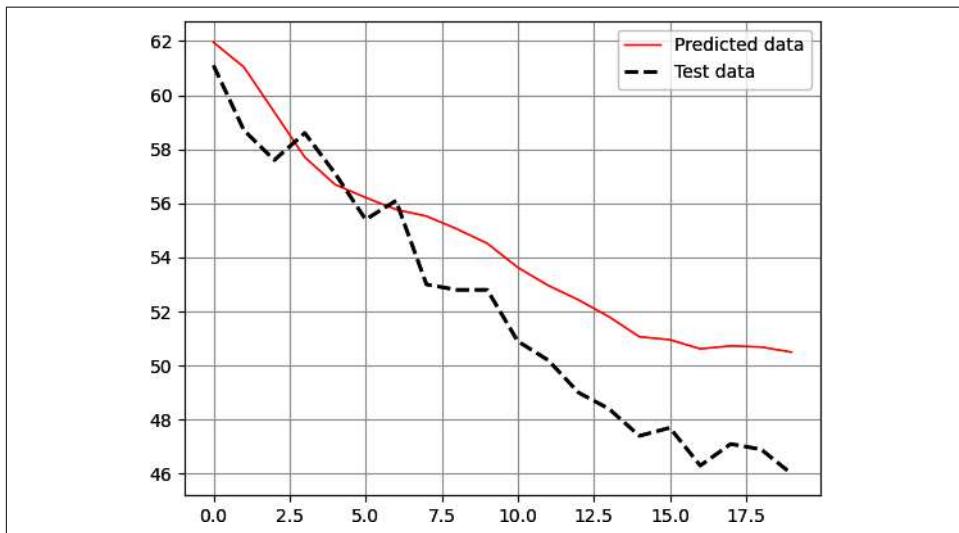


Figure 9-12. Multiperiod forecasts of model versus real values, with some optimization

The interpretation of the model at the time of the forecast was for a consecutive drop in the ISM PMI for 18 months. The model seems to have done a good job at predicting this direction. Note that you may get different results due to the random initialization of the algorithm, which may impact its convergence to a minimum loss function. You can use `random_state` to get the same results every time (you saw this in Chapter 7).



The ISM PMI has a positive correlation with the US gross domestic product (GDP) and a slight positive correlation with the S&P 500. To be more precise, bottoms in the ISM PMI have coincided with bottoms in the equity markets.

Out of curiosity, let's try running the model on very simple and basic hyperparameters:

```
num_lags = 1
train_test_split = 0.80
num_neurons_in_hidden_layers = 2
num_epochs = 10
batch_size = 1
forecast_horizon = 18
```

Obviously, with one lag, the model will only take into account the previous value to learn how to predict the future. The hidden layers will only contain two neurons each and will run for only 10 epochs using a batch size of 1. Naturally, you would not expect satisfying results by using these hyperparameters. [Figure 9-13](#) compares the predicted values to the real values. Notice the huge discrepancy as the model does not pick on the magnitude or the direction.

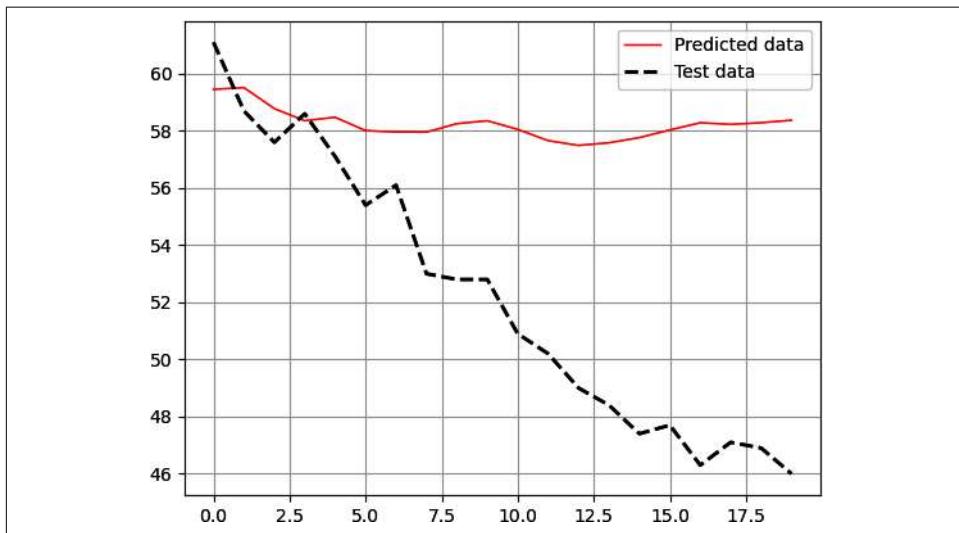


Figure 9-13. Multiperiod forecasts of the model versus real values, using basic hyperparameters

This is why hyperparameter optimization is important and a certain degree of complexity is needed. After all, these time series are not simple and carry a significant amount of noise in them.

Finally, let's have a look at the results of running the following hyperparameters on Basel's temperature data, as you saw at the beginning of this section:

```
num_lags = 500
train_test_split = 0.80
num_neurons_in_hidden_layers = 128
num_epochs = 50
batch_size = 12
forecast_horizon = 500
```

Figure 9-14 compares the predicted values to the real values using the temperature time series. The number of predicted observations is 500.

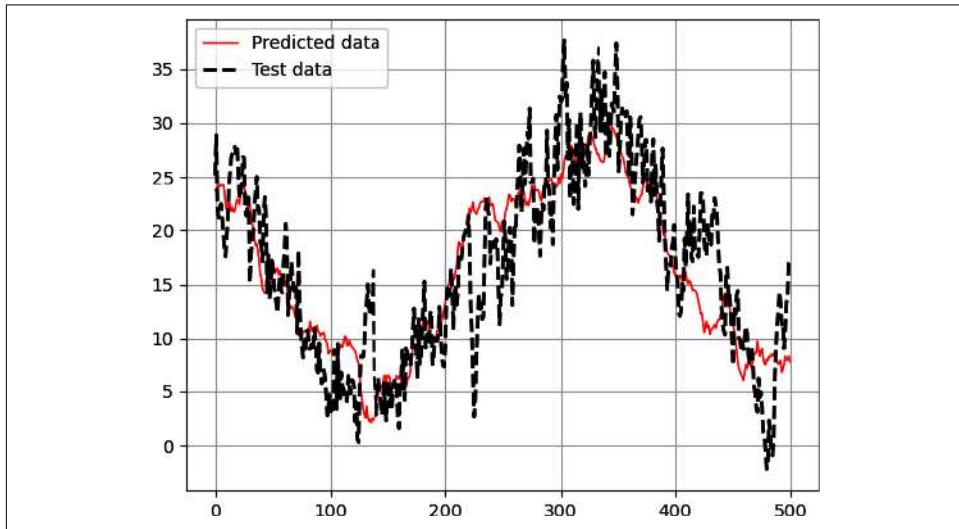


Figure 9-14. Multiperiod forecasts of the model versus real values, using the temperature time series

Which prediction technique to use depends on your preferences and needs. It is worth mentioning an additional MPF technique referred to as the *multioutput model*, which is a one-shot forecast of a number of values. This means that the model is trained over the training set with the aim of producing an instant predefined number of outputs (predictions). Obviously, this model may be computationally expensive and would require a sizable amount of data.

Applying Regularization to MLPs

Chapter 8 discussed two regularization concepts regarding deep learning:

- Dropout as a regularization technique that randomly deactivates neurons during training to prevent overfitting
- Early stopping as a method to prevent overfitting by monitoring the model's performance and stopping training when performance starts to degrade

Another regularization technique worth discussing is *batch normalization*, a technique used in deep learning to improve the training and generalization of neural networks. It normalizes the inputs of each layer within a mini batch during training, which helps in stabilizing and accelerating the learning process.

The main idea behind batch normalization is to ensure that the inputs to a layer have zero mean and unit variance. This normalization is applied independently to each feature (or neuron) within the layer. The process can be summarized in the following steps:

1. For each feature in a mini batch, calculate the mean and variance across all the samples in the batch.
2. Subtract the mean and divide by the standard deviation (the square root of the variance) for each feature.
3. After normalization, the features are scaled and shifted by learnable parameters. These parameters allow the model to learn the optimal scale and shift for each normalized feature.

This section presents a simple forecasting task using LSTMs with the addition of the three regularization techniques. The time series is the S&P 500's 20-day rolling auto-correlation data. Import the required libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
import pandas_datareader as pdr
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
```

Import and preprocess the data:

```
# Set the start and end dates for the data
start_date = '1990-01-01'
end_date   = '2023-06-01'
# Fetch S&P 500 price data
data = np.array((pdr.get_data_fred('SP500', start = start_date,
                                    end = end_date)).dropna())
```

Calculate the 20-day autocorrelation of the close prices:

```
rolling_autocorr = pd.DataFrame(data).rolling(window =
                                         20).apply(lambda x:
                                         x.autocorr(lag=1)).dropna()
rolling_autocorr = np.reshape(np.array(rolling_autocorr), (-1))
```



In Python, a `lambda` function, also known as an *anonymous* function, is a small, unnamed function that can have any number of arguments but can only have one expression. These functions are often used for creating simple, inline functions without needing to define a full function using the `def` keyword. Here's a simple example to illustrate how `lambda` works:

```
# Create an anonymous function to divide two variables
divide = lambda x, y: x / y
# Call the function
result = divide(10, 2)
```

The output will be the float 5.0 stored in `result`.

The `apply()` function is a method that is available in *pandas*. It is primarily used to apply a given function along an axis of a dataframe.

Before continuing, try plotting the S&P 500 price data versus its 20-day autocorrelation that you just calculated. Use this code to generate [Figure 9-15](#):

```
fig, axes = plt.subplots(nrows = 2, ncols = 1)
axes[0].plot(data[-350:], label = 'S&P 500', linewidth = 1.5)
axes[1].plot(rolling_autocorr[-350:], label = '20-Day Autocorrelation',
            color = 'orange', linewidth = 1.5)
axes[0].legend()
axes[1].legend()
axes[0].grid()
axes[1].grid()
axes[1].axhline(y = 0.95, color = 'black', linestyle = 'dashed')
```

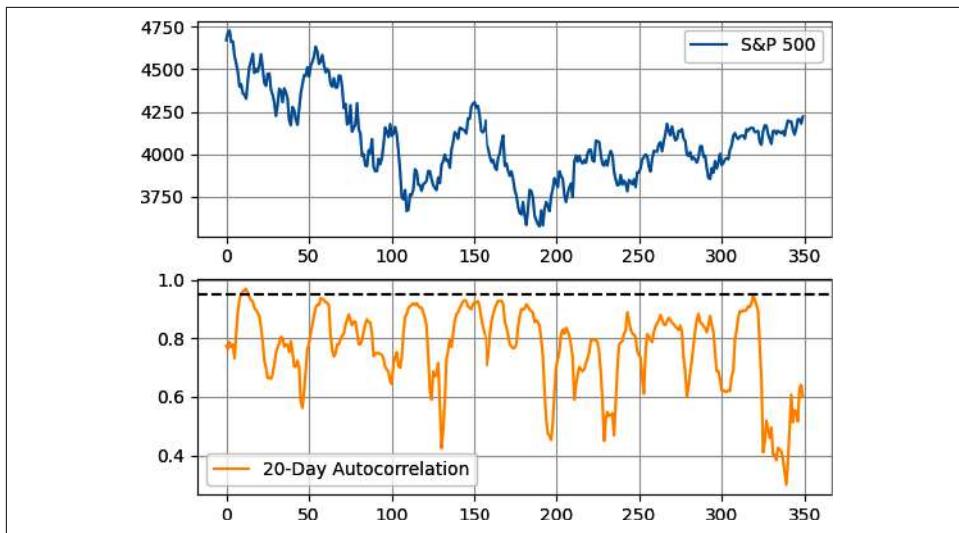


Figure 9-15. The S&P 500 versus its 20-day price autocorrelation (lag = 1)

What you should retain from the chart and from the intuition of autocorrelation is that whenever autocorrelation approaches 1.00, the current trend may break, thus leading to a market correction. This is not a perfect assumption, but you can follow these basic rules to interpret the rolling autocorrelation observations:

- A trending market (bullish or bearish) will have its autocorrelation approach 1.00 sooner or later. When this happens, it may signal a pause in the underlying trend, or in rarer occasions, a full reversal.
- A sideways (ranging) market will have a low autocorrelation. If the autocorrelation approaches historical lows, then it may mean that the market is ready to trend.

Let's now continue building the algorithm. The next step is to set the hyperparameters and prepare the arrays:

```

num_lags = 500
train_test_split = 0.80
num_neurons_in_hidden_layers = 128
num_epochs = 100
batch_size = 20
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(rolling_autocorr,
                                                      num_lags,
                                                      train_test_split)

```

Transform the input arrays into three-dimensional structures so that they are processed into the LSTM architecture with no issues:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Design the LSTM architecture and add the dropout layer and batch normalization. Add the early stopping implementation while setting `restore_best_weights` to `True` so as to keep the best parameters for the prediction over the test data:

```
# Create the LSTM model
model = Sequential()
model.add(LSTM(units = num_neurons_in_hidden_layers, input_shape =
               (num_lags, 1)))
# Adding batch normalization and a dropout of 10%
model.add(BatchNormalization())
model.add(Dropout(0.1))
# Adding the output layer
model.add(Dense(units = 1))
# Compile the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Early stopping implementation
early_stopping = EarlyStopping(monitor = 'loss', patience = 15,
                               restore_best_weights = True)
# Train the model
model.fit(x_train, y_train, epochs = num_epochs,
           batch_size = batch_size, callbacks = [early_stopping])
```

Predict and plot the results:

```
# Predicting in-sample
y_predicted_train = np.reshape(model.predict(x_train), (-1, 1))
# Predicting out-of-sample
y_predicted = np.reshape(model.predict(x_test), (-1, 1))
# Plotting
plot_train_test_values(300, 50, y_train, y_test, y_predicted)
```

Figure 9-16 shows the predictions versus the real values. The model has stopped the training before reaching 100 epochs due to the callback from the early stopping mechanism.

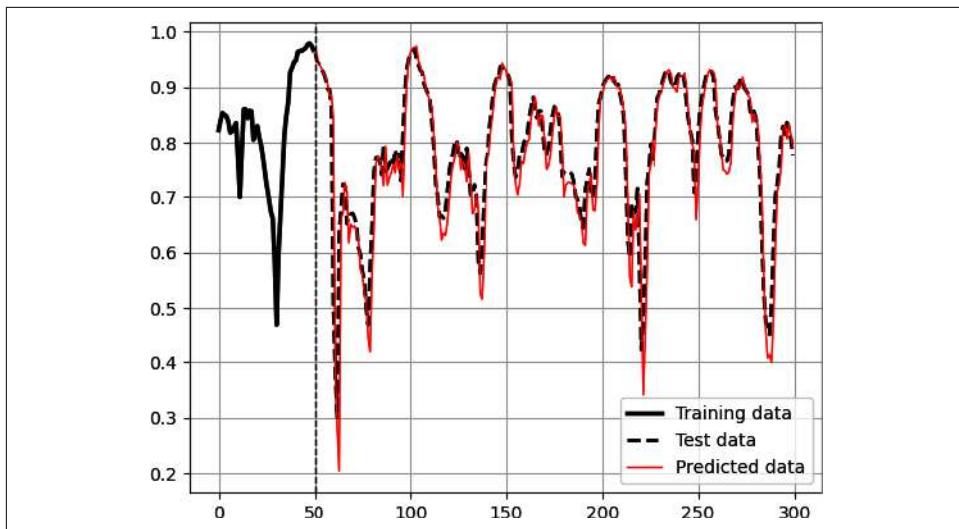


Figure 9-16. Predicting correlation

The results are as follows:

```

Accuracy Train = 70.37 %
Accuracy Test = 68.12 %
RMSE Train = 0.0658945761
RMSE Test = 0.0585669847
Correlation In-Sample Predicted/Train = 0.945
Correlation Out-of-Sample Predicted/Test = 0.936

```

It's important to note that using indicators such as rolling autocorrelation should be done with caution. They provide insights into historical patterns, but they don't guarantee future performance. Additionally, the effectiveness of rolling autocorrelation as a technical indicator depends on the nature of the data and the context in which it's being used. You can try applying the MPF method on the autocorrelation data.

Other regularization techniques that exist include the following:

L1 and L2 regularization

Also known as *weight decay*, L1 and L2 regularization add a penalty term to the loss function based on the magnitude of the weights. *L1 regularization* adds the absolute values of the weights to the loss, which encourages sparsity in the model. *L2 regularization* adds the squared values of the weights, which discourages large weight values and tends to distribute the influence of features more evenly.

DropConnect

This technique is similar to dropout but is applied to connections rather than neurons. This technique randomly drops connections between layers during training.

Weight constraints

Limiting the magnitude of weight values can prevent the model from learning complex patterns from noise and helps regularize the model.

Adversarial training

Training the model using adversarial examples can improve its robustness by making it more resistant to small perturbations in the input data.

Using these regularization techniques doesn't guarantee a better result than using the model without them. However, deep learning best practices encourage such techniques to avoid more serious problems like overfitting.



When manually uploading an Excel file (using *pandas*, for example) that contains historical data, make sure that it has a shape of $(n,)$ and not a shape of $(n, 1)$. This ensures that when you use the `data_preprocessing()` function, the four training/test arrays will be created with the proper dimensions.

To transform an $(n, 1)$ array to $(n,)$, use the following syntax:

```
data = np.reshape(data, (-1))
```

To transform an $(n,)$ array to $(n, 1)$, use the following syntax:

```
data = np.reshape(data, (-1, 1))
```

Summary

This chapter presented a few techniques that may improve the different machine and deep learning algorithms. I like to refer to such techniques as *satellites* since they hover around the main component, that is, neural networks. Optimizations and enhancements are crucial to the success of the analysis. For example, some markets may benefit from the forecasting threshold technique and fractional differentiation. Trial and error is key to understanding your data, and as you begin [Chapter 10](#) and learn about reinforcement learning, you will see that trial and error is not just a human task. It can also be a computer task.

Deep Reinforcement Learning for Time Series Prediction

Reinforcement learning is a branch of machine learning that deals with sequential decision-making problems. Algorithms in this branch learn to make optimal decisions by interacting with an environment and receiving feedback in the form of rewards. In the context of time series forecasting, it can be used to develop models that make sequential predictions based on historical data. Traditional forecasting approaches often rely on statistical methods or supervised learning techniques, which assume independence between data points. However, time series data exhibits temporal dependencies and patterns, which may be effectively captured using reinforcement learning.

Reinforcement learning models for time series forecasting typically involve an agent that takes actions based on observed states and receives rewards based on the accuracy of its predictions. The agent learns through trial and error to maximize cumulative rewards over time. The key challenge is finding an optimal balance between *exploration* (trying out new actions) and *exploitation* (using learned knowledge).

This chapter gives a basic overview of reinforcement learning and deep reinforcement learning with regard to predicting time series data.

Intuition of Reinforcement Learning

Simplification is always the right path toward understanding more advanced details. So let's look at reinforcement learning from a simple point of view before digging deeper.

Reinforcement learning deals primarily with rewards and penalties. Imagine a child who gets a reward for doing good things and a punishment for doing bad things. Over time, that child will grow and will develop their experience so that they do good things and try to avoid doing bad things as much as possible (no one is perfect). Therefore, the learning is done through experience.

From a time series perspective, the main idea is the same. Imagine training a model on past data and letting it then learn by experience, rewarding it for good predictions and calibrating its parameters when it makes a mistake so that it can achieve better accuracy next time. The algorithm is greedy in nature and wants to maximize its rewards; therefore, over time it becomes better at predicting the next likely value, which is of course dependent on the quality and the signal-to-noise ratio of the analyzed time series.

The term *reinforcement learning* comes from the fact that *positive reinforcement* is given to the algorithm when it makes right decisions and *negative reinforcement* is given when it makes bad decisions. The first three concepts you must know are states, actions, and rewards:

States

The features at every time step. For example, at a certain time step, the current state of the market is its OHLC data and its volume data. In more familiar words, states are the explanatory variables.

Actions

The decisions a trader may make at every time step. They generally involve buying, selling, or holding. In more familiar words, actions are the algorithms' decisions when faced with certain states (a simple discretionary example of this would be a trader noticing an overvalued market and deciding to initiate a buy order).

Rewards

The results of correct actions. The simplest reward is the positive return. Note that a poorly designed reward function can lead to model issues such as a buy-and-hold strategy.¹

¹ A buy-and-hold strategy is a passive action whereby the trader or the algorithm initiates one buy order and holds it for a long time in an attempt to replicate the market's return and minimize transaction costs incurred from excessive trading.

Table 10-1 shows the three main elements of reinforcement learning.

Table 10-1. A hypothetical decision table

	Time	Open	High	Low	Close		Action	Reward
States	1	10	14	8	10		BUY	0
States	2	10	15	6	13		BUY	3
States	3	13	16	8	14		SELL	-1
States	4	10	16	8	14		HOLD	0

States are the rows that go from the Time column to the Close column. Actions can be categorical, as you can see from the Action column, and Rewards can either be numerical (e.g., a positive or negative profit) or categorical (e.g., profit or loss label).

From the preceding list, it seems complicated to just design a system that looks for rewards. A *reward function* quantifies the desirability or utility of being in a particular state or taking a specific action. The reward function therefore provides feedback to the agent, indicating the immediate quality of its actions and guiding its learning process. Before we discuss reward functions in more detail, let's look at what a state-action table is (also known as a Q-table).

A *Q-table*, short for *quality table*, is a data structure to store and update the expected value (called the *Q-value*) of taking a particular action in a given state. The Q-value of a state-action pair (s, a) at time t represents the expected cumulative reward that an agent can achieve by taking action a in state s following a specific policy. The Q-table is therefore a table-like structure that maps each state-action pair to its corresponding Q-value.

Initially, the Q-table is usually initialized with arbitrary values or set to zero. As the algorithm explores the environment (market) and receives rewards, it updates the Q-values in the table based on the observed rewards and the estimated future rewards. This process is typically done using an algorithm such as Q-learning.



Over time, through repeated exploration and exploitation, the Q-table gradually converges to more accurate estimates of the optimal Q-values, representing the best actions to take in each state. By using the Q-table, the agent can make informed decisions and learn to maximize its cumulative rewards in the given environment. Remember, a reward can be a profit, Sharpe ratio, or any other performance metric.

Q-learning is a popular reinforcement learning algorithm that enables an agent to learn optimal actions by iteratively updating its action-value function, known as the *Bellman equation*, defined as follows:

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma \max[Q(s_{t+1}, a_{t+1})]$$

$Q(s_t, a_t)$ is the expected reward

$R(s_t, a_t)$ is the reward table

γ is the learning rate (known as *gamma*)

The larger the learning rate (γ), the more the algorithm takes into account the previous experiences. Notice that if γ is equal to zero, it would be synonymous to learning nothing as the second term will cancel itself out. As a simple example, consider Table 10-2.

Table 10-2. R-table

Time (State)	Act (Action)	Wait (Action)
1	2 reward units	0 reward units
2	2 reward units	0 reward units
3	2 reward units	0 reward units
4	2 reward units	0 reward units
5	2 reward units	4 reward units

The table describes the results of actions through time. At every time step, acting (doing something) will give a reward of 2, while waiting to act on the fifth time step will give a reward of 4. This means that the agent can make one of the following choices:

- Act now and get 2 reward units.
- Wait before acting and get 4 reward units.

Let's assume $\gamma = 0.80$. Using the Bellman equation and working backward will get you the following results:

$$Q(s_1, a_1) = 0 + 0.8(2.04) = 1.63$$

$$Q(s_2, a_2) = 0 + 0.8(2.56) = 2.04$$

$$Q(s_3, a_3) = 0 + 0.8(3.20) = 2.56$$

$$Q(s_4, a_4) = 0 + 0.8(4.00) = 3.20$$

Table 10-2 may be updated to become **Table 10-3** as follows.

Table 10-3. Q-table

Time (State)	Act (Action)	Wait (Action)
1	2 reward units	1.63 reward units
2	2 reward units	2.04 reward units
3	2 reward units	2.56 reward units
4	2 reward units	3.20 reward units
5	2 reward units	4.00 reward units

Therefore, the Q-table is continuously updated with the implied rewards to help maximize the final reward. To understand why the term *max* is in the Bellman equation, consider the example in **Table 10-4**.

Table 10-4. R-table

Time (State)	Buy	Sell	Hold
1	5	8	8
2	3	2	1
3	2	5	6

Calculate the would-be value of x in a Q-table (**Table 10-5**) assuming a learning rate of 0.4.

Table 10-5. Q-table

Time (State)	Buy	Sell	Hold
1	?	?	?
2	?	x	?
3	2	5	6

Following the formula, you should get this result:

$$x = 2 + 0.4(\max(2, 5, 6)) = 4.4$$

States (features) must be predictive in nature so that the reinforcement learning algorithm predicts the next value with an accuracy better than random. Examples of features can be the values of the relative strength index (RSI), moving averages, and lagged close prices.

It is crucial to keep in mind that the inputs' statistical preference remains the same, that is, stationary. This begs the question: how are moving averages used as inputs if they are not stationary? The simple answer is through the usual transformation, which is to take the percentage difference.



It is possible to use fractional differencing to transform a nonstationary time series into a stationary one while retaining its memory.

A *policy* defines the behavior of an agent in an environment. It is a mapping from states to actions, indicating what action the agent should take in a given state. The policy essentially guides the agent's decision-making process by specifying the action to be executed based on the observed state.

The goal of reinforcement learning is to find an optimal policy that maximizes the agent's long-term cumulative reward. This is typically achieved through a trial-and-error process, where the agent interacts with the environment, takes actions, receives rewards, and adjusts its policy based on the observed outcomes.

The exploitation policy is generally faster than the exploration policy but may be more limited as it seeks a greater and immediate reward, while there may be a path afterward that leads to an even greater reward. Ideally, the best policy to take is a combination of both. But how do you determine this optimal mix? That question is answered by epsilon (ϵ).

Epsilon is a parameter used in exploration–exploitation trade-offs. It determines the probability with which an agent selects a random action (exploration) versus selecting the action with the highest estimated value (exploitation).

Commonly used exploration strategies include epsilon-greedy and softmax. In *epsilon-greedy*, the agent selects the action with the highest estimated value with a probability of $(1 - \epsilon)$, and then it selects a random action with a probability of ϵ . This allows the agent to explore different actions and potentially discover better policies. As the agent learns over time, the epsilon value is often decayed gradually to reduce exploration and focus more on exploitation.² *Softmax* action selection considers the estimated action values but introduces stochasticity in the decision-making process. The temperature parameter associated with softmax determines the randomness in action selection, where a higher temperature leads to more exploration.

² Keep epsilon decay in mind as it will be used as a variable in the code later.



Do not mix up epsilon and gamma:

- *Gamma* is a parameter that determines the importance of future rewards. It controls the extent to which the agent values immediate rewards compared to delayed rewards (hence, it is related to a delayed gratification issue). The value of gamma is typically a number between 0 and 1, where a value closer to 1 means the agent considers future rewards more heavily, while a value closer to 0 gives less importance to future rewards. To understand this more, consider having another look at the Bellman equation.
- *Epsilon* is a parameter used in exploration–exploitation trade-offs. It determines the probability with which an agent selects a random action (exploration) versus selecting the action with the highest estimated value (exploitation).

At this point, you may feel overwhelmed by the amount of new information presented, especially because it differs from what you have seen so far in the book. Before moving to the more complex deep reinforcement learning discussion, a quick summary of what you have seen in this chapter until now may be beneficial. Reinforcement learning is essentially giving the machine a task that it will then learn how to do on its own.

With time series analysis, states represent the current situation or condition of the environment at a particular time. An example of state is a technical indicator's value. States are represented by Q-tables. Actions are self-explanatory and can be buy, sell, or hold (or even a more complex combination such as decrease weight and increase weight). Rewards are what the algorithm is trying to maximize and can be profit per trade, Sharpe ratio, or any sort of performance evaluation metric. A reward can also be a penalty such as the number of trades or maximum drawdown (in such a case, you are aiming to minimize it). The reinforcement learning algorithm will go through many iterations and variables through different policies to try to detect hidden patterns and optimize trading decision so that profitability is maximized. This is easier said than done (or coded).

One question is begging an answer: is using a Q-table to represent the different states of financial time series efficient? This question is answered in the next section.

Deep Reinforcement Learning

Deep reinforcement learning combines reinforcement learning techniques with deep learning architectures, particularly deep neural networks. It involves training agents to learn optimal behavior and make decisions by interacting with an environment, using deep neural networks to approximate value functions or policies.

The main difference between a reinforcement learning algorithm and a deep reinforcement learning algorithm is that the former estimates Q-values using the Q-table, while the latter estimates Q-values using ANNs (see [Chapter 8](#) for details on artificial neural networks).



As a reminder, *artificial neural networks* (ANNs) are a type of computational model inspired by the structure and functioning of the human brain. A neural network consists of interconnected nodes organized into layers. The three main types of layers are the input layer, hidden layers, and the output layer. The input layer receives the initial data, which is then processed through the hidden layers, and finally, the output layer produces the network's prediction.

The main objective of this section is to understand and design a deep reinforcement learning algorithm with the aim of data prediction. Keep in mind that reinforcement learning is still not heavily applied since it suffers from a few issues (discussed at the end of this section) that need to be resolved before making it one of the main trading algorithms in quantitative finance.

Therefore, deep reinforcement learning will have two main elements with important tasks:

- A deep neural network architecture to recognize patterns and approximate the best function that relates dependent and independent variables
- A reinforcement learning architecture that allows the algorithm to learn by trial and error how to maximize a certain profit function

Let's continue defining a few key concepts before putting things together. *Replay memory*, also known as *experience replay*, involves storing and reusing past experiences to enhance the learning process and improve the stability and efficiency of the training.

In deep reinforcement learning, an agent interacts with an environment, observes states, takes actions, and receives rewards. Each observation, action, reward, and resulting next state is considered an experience. The replay memory serves as a buffer that stores a collection of these experiences.

The replay memory has the following key features:

Storage

The replay memory is a data structure that can store a fixed number of experiences. Each experience typically consists of the current state, the action taken, the resulting reward, the next state, and a flag indicating whether the episode terminated.

Sampling

During the training process, instead of using experiences immediately as they occur, the agent samples a batch of experiences from the replay memory. Randomly sampling experiences from a large pool of stored transitions helps in decorrelating the data and breaking the temporal dependencies that exist in consecutive experiences.

Batch learning

The sampled batch of experiences is then used to update the agent's neural network. By learning from a batch of experiences rather than individual experiences, the agent can make more efficient use of computation and improve the learning stability. Batch learning also allows for the application of optimization techniques, such as stochastic gradient descent, to update the network weights.

The replay memory provides several benefits to deep reinforcement learning algorithms. Among those benefits is experience reuse, as the agent can learn from a more diverse set of data, reducing the bias that can arise from sequential updates. Breaking correlations is another benefit since the sequential nature of experience collection in reinforcement learning can introduce correlations between consecutive experiences. Randomly sampling experiences from the replay memory helps break these correlations, making the learning process more stable.

So far, we have discussed the following steps:

1. Defining and initializing the environment
2. Designing the neural network architecture
3. Designing the reinforcement learning architecture with experience replay to stabilize the learning process
4. Interacting with the environment and storing experiences until the learning process is done and predictions on new data are done

One thing we have not discussed is how to reduce overestimations, which can be achieved by doubling down on the neural network architecture.

The *Double Deep Q-Network* (DDQN) model is an extension of the original DQN architecture introduced by DeepMind in 2015. The primary motivation behind

DDQN is to address a known issue in the DQN algorithm called *overestimation bias*, which can lead to suboptimal action selection.

In the original DQN, the action values (Q-values) for each state-action pair are estimated using a single neural network. However, during the learning process, the Q-values are estimated using the maximum Q-value among all possible actions in the next state (take a look at [Table 10-5](#)). This maximum Q-value can sometimes result in an overestimation of the true action values, leading to a suboptimal policy.

The DDQN addresses this overestimation bias by utilizing two separate neural networks: the Q-network and the target-network. The *Q-network* is a deep neural network that approximates the action-value function (Q-function). In other words, it estimates the value of each possible action in a given state. The Q-network's parameters (weights and biases) are learned through training to minimize the difference between predicted Q-values and target Q-values. The *target network* is a separate copy of the Q-network that is used to estimate the target Q-values during training. It helps stabilize the learning process and improve the convergence of the Q-network. The weights of the target network are not updated during training; instead, they are periodically updated to match the weights of the Q-network.

The key idea behind the DDQN is to decouple the selection of actions from the estimation of their values.



The algorithm updates the Q-network regularly and the target network occasionally. This is done to avoid the issue of the same model being used to estimate the Q-value from the next state and then giving it to the Bellman equation to estimate the Q-value for the current state.

So, to put these elements into an ordered sequence, here's how the deep reinforcement learning architecture may look:

1. The environment is initialized.
2. The epsilon value is selected. Remember, epsilon is the exploration-exploitation trade-off parameter used to control the agent's behavior during training.
3. The current state is fetched. Remember, an example of the current state may be the OHLC data, the RSI, the standard deviation of the returns, or even the day of the week.
4. In the first round, the algorithm selects the action through exploration as the model is not trained yet; therefore, the action is randomly selected (e.g., from a choice panel of buy, sell, and hold). If it's not the first step, then exploitation may be used to select the action. Exploitation is where the action is determined by the neural network model.

5. The action is applied.
6. The previous elements are stored in replay memory.
7. The inputs and the target array are fetched and the Q-network is trained.
8. If the round is not over, repeat the process starting at step 3. Otherwise, train the target network and repeat from step 1.

To illustrate the algorithm, let's use it on the synthetic sine wave time series. Create the time series and then apply the deep reinforcement learning algorithm with the aim of predicting the future values.

The full code can be found in the [GitHub repository](#) (for replication purposes).

[Figure 10-1](#) shows the test data (solid line) versus the predicted data (dashed line) using 1 epoch, 5 inputs (lagged values), a batch size of 64, and 1 hidden layer with 6 neurons.

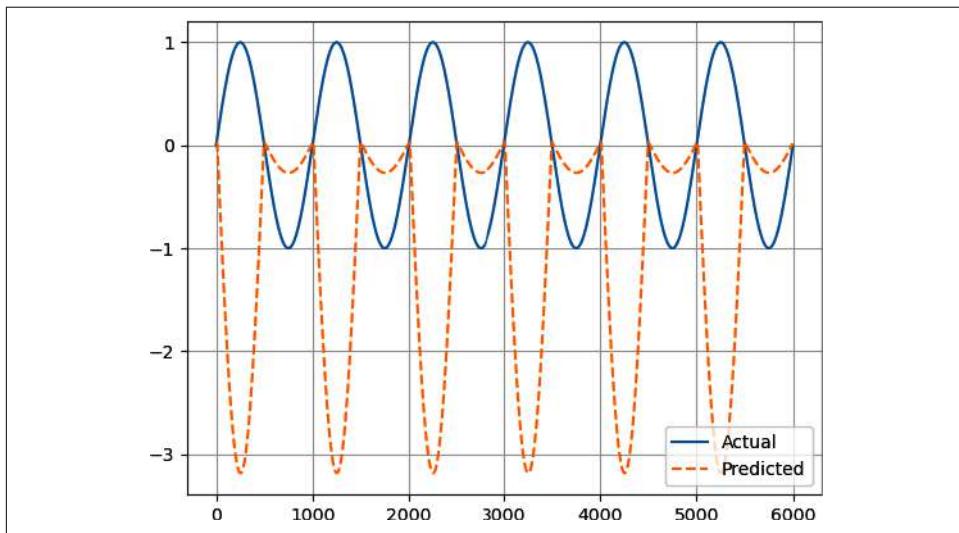


Figure 10-1. Test data versus predicted data using 1 epoch, 5 inputs, a batch size of 64, and 1 hidden layer with 6 neurons

[Figure 10-2](#) shows the test data (solid line) versus the predicted data (dashed line) using 1 epoch, 5 inputs (lagged values), a batch size of 64, and 2 hidden layers with each having 6 neurons.

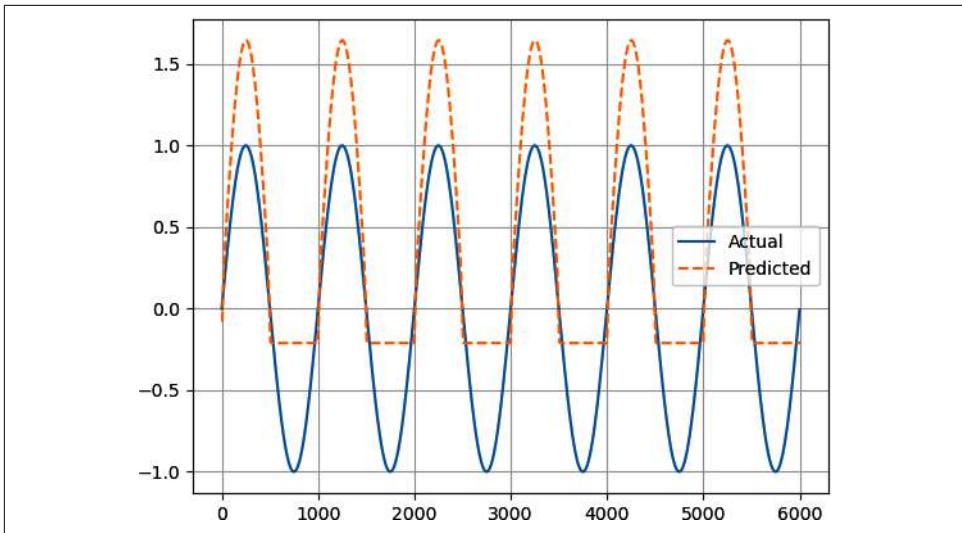


Figure 10-2. Predicted values versus actual values using 1 epoch, 5 inputs, a batch size of 64, and 2 hidden layers with each having 6 neurons

Figure 10-3 shows the predictions using 10 epochs, 5 inputs (lagged values), a batch size of 32, and 2 hidden layers with each having 6 neurons.

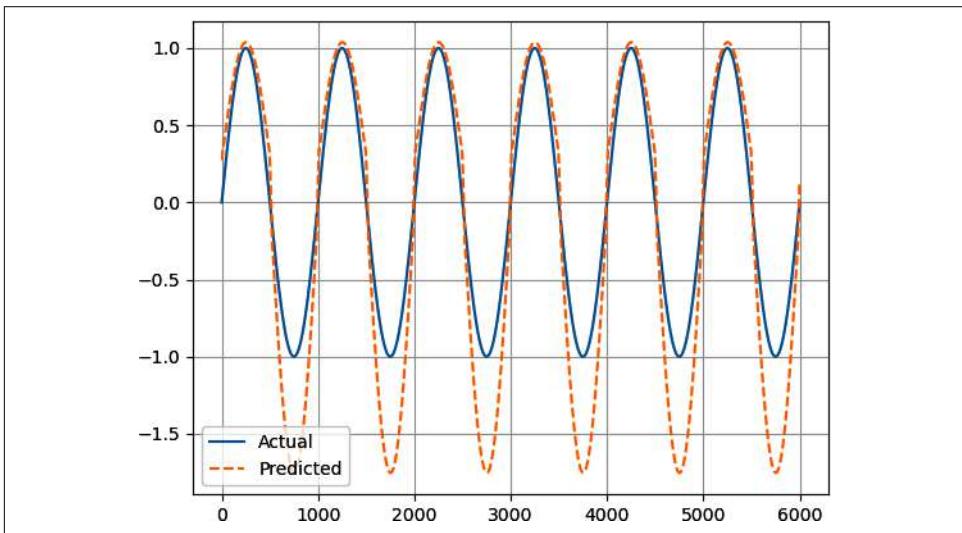


Figure 10-3. Predicted values versus actual values using 10 epochs, 5 inputs, a batch size of 32, and 2 hidden layers with each having 6 neurons

Figure 10-4 shows the predictions using 10 epochs, 5 inputs (lagged values), a batch size of 32, and 2 hidden layers with each having 24 neurons.

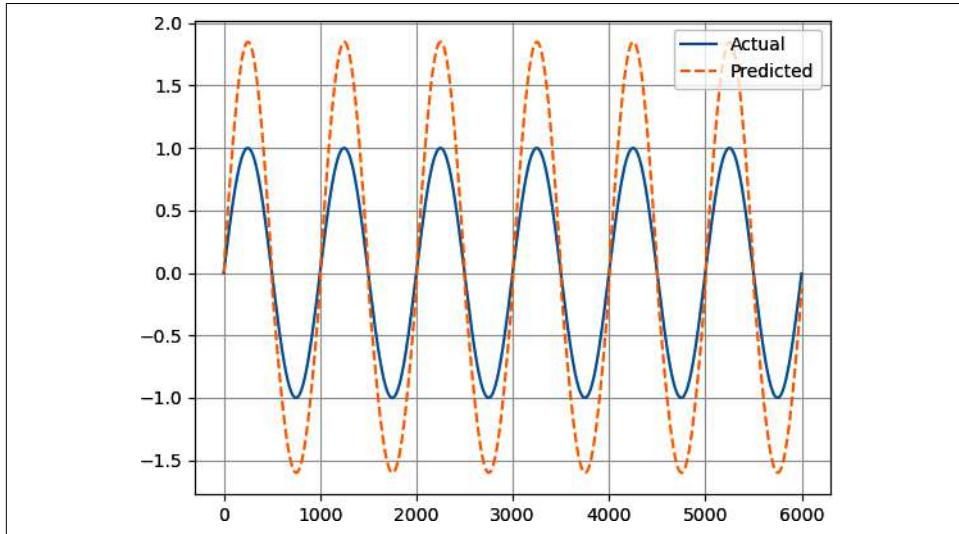


Figure 10-4. Predicted values versus actual values using 10 epochs, 5 inputs, a batch size of 32, and 2 hidden layers with each having 24 neurons

Figure 10-5 shows the predictions using 10 epochs, 8 inputs (lagged values), a batch size of 32, and 2 hidden layers with each having 64 neurons.

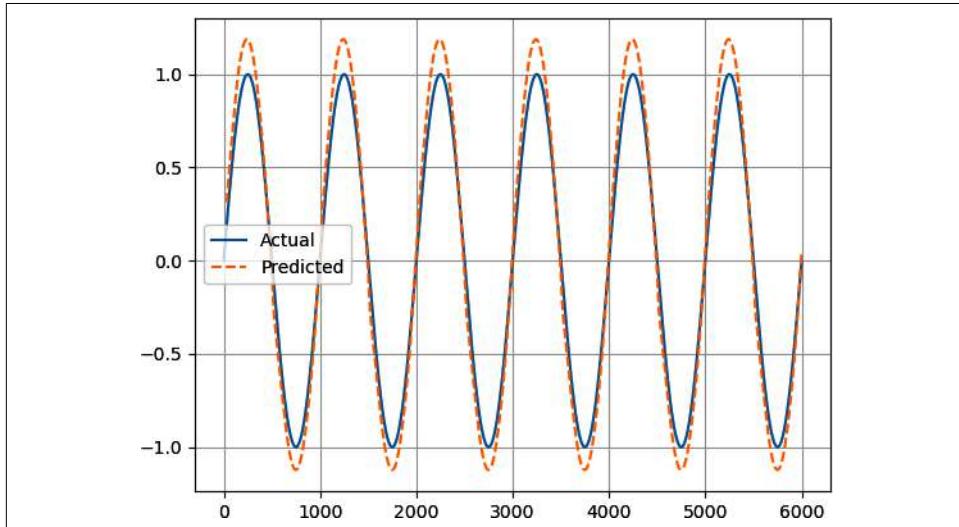


Figure 10-5. Predicted values versus actual values using 10 epochs, 8 inputs, a batch size of 32, and 2 hidden layers with each having 64 neurons

As you know, the more epochs, the better the fit—up to a certain point, where overfitting may start to become an issue. Fortunately, by now you know how to reduce that risk.



Note that the sine wave example is a very basic one, and more complex data can be used with the algorithm. The choice of the sine wave time series is for illustrative purposes only, and you must use more sophisticated methods on more complex time series to be able to judge the algorithm.

Reinforcement learning is easily overfit and is more likely to learn simple patterns and not hidden and complicated ones. Also, you should now be aware of the difficulty of reward function design and choice of features. Furthermore, such models are often considered mystery boxes, making it difficult to explain the reasoning behind their predictions. All of these issues are now a barrier to implementing a stable and profitable deep reinforcement learning algorithm for trading.

Summary

Reinforcement learning can be applied to time series prediction tasks, where the goal is to make predictions about future values based on historical data. In this approach, an agent interacts with an environment representing the time series data. The agent receives observations of past values and takes actions to predict future values. The agent's actions involve adjusting its internal model or parameters to make predictions. It uses reinforcement learning algorithms to learn from past experiences and improve its prediction accuracy over time.

The agent receives rewards or penalties based on the accuracy of its predictions. Rewards can be designed to reflect the prediction error or the utility of the predictions for the specific application. Through a process of trial and error, the agent learns to associate certain patterns or features in the time series data with future outcomes. It learns to make predictions that maximize rewards and minimize errors.

The reinforcement learning process involves a balance between exploration and exploitation. The agent explores different prediction strategies, trying to discover patterns and make accurate predictions. It also exploits its existing knowledge to make predictions based on what it has learned so far. The goal of reinforcement learning for time series prediction is to train the agent to make accurate and reliable predictions. By continually receiving feedback and updating its prediction strategies, the agent adapts to changing patterns in the time series and improves its forecasting abilities.

Chapter 11 will show how to employ more deep learning techniques and applications.

Advanced Techniques and Strategies

By now, you should have a solid understanding of deep learning algorithms and how to develop a model to predict time series data. Even though this is just a first step toward deploying a profitable algorithm, you should know that you have come a long way since the beginning of the book. This chapter is divided into independent sections that discuss interesting ways of applying a few advanced deep learning techniques and methods for time series prediction and to enhance the process.

Using COT Data to Predict Long-Term Trends

The *Commitments of Traders* (COT) report is a [weekly publication](#) released by the US Commodity Futures Trading Commission (CFTC). It provides information on the positions held by various market participants in futures markets. The report is based on data collected from futures exchanges, including the Chicago Mercantile Exchange (CME) and the Intercontinental Exchange (ICE). The COT report categorizes traders into three main groups:

Commercial traders (also referred to as dealers or hedgers)

These are typically companies that use the futures market to hedge their main business activities. For example, a grain producer may use futures contracts to protect against price fluctuations in the agricultural market. Their positions are generally negatively correlated to the underlying market.

Noncommercial traders (also referred to as funds or leveraged money)

This group consists of large speculators, such as hedge funds and commodity trading advisors. Noncommercial traders often take positions based on their market outlook and profit-seeking strategies. Their positions are generally positively correlated to the underlying market as they have a trend-following nature.

Nonreportable traders

This category includes small speculators and traders whose positions do not meet the reporting requirements set by the CFTC. They do not have a clear correlation with the underlying market.

The report provides a breakdown of the positions held by each group, indicating whether they are *net long* (holding more long positions than short positions) or *net short* (holding more short positions than long positions) in a particular futures market.

Traders and investors analyze the COT report to gain insights into the sentiment and behavior of different market participants. By monitoring changes in positions, they attempt to identify potential trends or reversals in the market. The report is especially popular in the commodity and currency markets, where it is used as a tool for fundamental analysis and to gauge market sentiment. This section covers how to do the following:

- Create an algorithm to download the COT data automatically and analyze it.
- Chart and understand the correlations between the COT values and their respective underlying markets. Additionally, check for stationarity in the COT values to see if they can be used directly in the algorithms.
- Create an LSTM algorithm to forecast the next COT value using lagged values and evaluate it. This will be referred to as the *indirect one-step COT model*.
- Create an LSTM algorithm to forecast a few weeks' worth of COT observations using the direct method. This will be referred to as the *MPF COT direct model*.
- Create an LSTM algorithm to forecast a few weeks' worth of COT observations using the recursive method. This will be referred to as the *MPF COT recursive model*.

There are mainly four columns of interest in the COT report: the long hedgers, the short hedgers, the long funds, and the short funds. They are the four basic pillars from which you calculate the net hedgers and the net funds. Some traders like to analyze every column before netting and seeing the bigger picture. It helps to understand the logic behind every column before proceeding to the netting process:

Long hedgers (the percentage long positions of commercial traders)

You can consider these hedgers *consumers* of the asset. A long hedger is an entity that buys futures on a certain asset (such as wheat) to hedge its main business. The primary goal is to protect itself from the risk of rising prices by securing a fixed price they need in the future. By doing so, they can plan their production costs more accurately and avoid potential losses if wheat prices increase. Therefore, long hedgers buy the asset in fear that it will go up. They usually buy it on its way down, which results in a negative correlation with the price of the asset.

Short hedgers (the percentage short positions of commercial traders)

You can consider these hedgers *producers* of the asset. A short hedger is an entity that sells short futures on a certain asset to hedge its main business. The primary goal is to protect itself from the risk of falling prices. Therefore, short hedgers sell in fear that it will go down. They usually sell it on its way up, which results in a positive correlation with the price of the asset (this means that the number of short hedgers rises as the price of the asset rises).

Long funds (the percentage long positions of noncommercial traders)

Speculative long fund positions are buyers of futures contracts in anticipation that the price will rise. They have a positive correlation with the price of the asset on account of their trend-following nature.

Short funds (the percentage short positions of noncommercial traders)

Speculative short fund positions are sellers of futures contracts in anticipation that the price will go down. They have a negative correlation with the price of the asset on account of their trend-following nature. An example of this would be the decreased number of short funds as the asset goes up.



Netting the COT report can be done in different ways depending on your needs. If you prefer to focus on commercial traders, then you can simply take the difference between commercial longs and commercial shorts (or as you may call them, consumers and producers). If you prefer to focus on noncommercial traders, then you can take the difference between noncommercial longs and non-commercial shorts. And if you prefer a global image, then you can take the difference between the netted commercial and noncommercial traders so that you're left with only one time series that summarizes the global picture of the market positioning on a certain asset.

Table 11-1 sheds some light on the calculation of net COT values.

Table 11-1. COT netting representation

Hedger long	Hedger short	Fund long	Fund short	Net hedger	Net fund	Net COT
A	B	C	D	E = A - B	F = C - D	G = F - E

The net COT value has the following formula:

$$\text{Net COT} = \text{Net funds} - \text{Net hedgers}$$

It shares a positive correlation with the underlying price of the asset. Figure 11-1 shows the net COT positioning on the Canadian dollar (CAD).

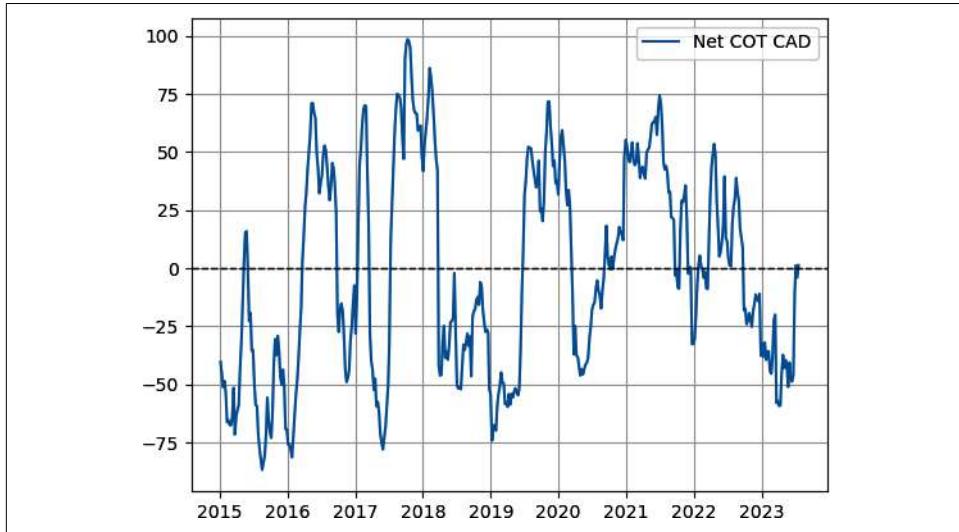


Figure 11-1. Net COT CAD since 2015; notice the mean-reverting nature of the values.

Let's see how to download COT values using Python. First, `pip install` the library that allows you to automatically download COT values from the CFTC website:

```
pip install cot_reports
```

Just in case the library has issues, you can use the predownloaded COT reports in Excel format found in the [GitHub repository](#) (a code block at the end of this section is provided for this manual import).

Import the required libraries to download the historical observations of the COT report. For simplicity, let's choose CAD positioning:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from master_function import import_cot_data
```

The `import_cot_data()` function that allows you to fetch the COT values of a selected market is defined as follows (found in `master_function.py`):

```
def import_cot_data(start_year, end_year, market):
    df = pd.DataFrame()
    for i in range(start_year, end_year + 1):
        single_year = pd.DataFrame(cot.cot_year(i,
                                                cot_report_type='traders_in_financial_futures_fut'))
        df = pd.concat([single_year, df], ignore_index=True)
    new_df = df.loc[:, ['Market_and_Exchange_Names',
```

```

'Report_Date_as_YYYY-MM-DD',
'Pct_of_OI_Dealer_Long_All',
'Pct_of_OI_Dealer_Short_All',
'Pct_of_OI_Lev_Money_Long_All',
'Pct_of_OI_Lev_Money_Short_All']]]

new_df['Report_Date_as_YYYY-MM-DD'] =
    pd.to_datetime(new_df['Report_Date_as_YYYY-MM-DD']))
new_df = new_df.sort_values(by='Report_Date_as_YYYY-MM-DD')
data = new_df[new_df['Market_and_Exchange_Names'] == market]
data['Net_COT'] = (data['Pct_of_OI_Lev_Money_Long_All'] - \
    data['Pct_of_OI_Lev_Money_Short_All']) - \
    (data['Pct_of_OI_Dealer_Long_All'] - \
    data['Pct_of_OI_Dealer_Short_All'])

return data

```

To import CAD COT values, use the following syntax:

```

CAD = 'CANADIAN DOLLAR - CHICAGO MERCANTILE EXCHANGE'
data = import_cot_data(2015, 2023, CAD)
data = np.array(data.iloc[:, -1], dtype = np.float64)

```

It is worth mentioning that other markets have the following code names that you can use to import them:

```

EUR = 'EURO FX - CHICAGO MERCANTILE EXCHANGE'
GBP = 'BRITISH POUND STERLING - CHICAGO MERCANTILE EXCHANGE'
JPY = 'JAPANESE YEN - CHICAGO MERCANTILE EXCHANGE'
CHF = 'SWISS FRANC - CHICAGO MERCANTILE EXCHANGE'
AUD = 'AUSTRALIAN DOLLAR - CHICAGO MERCANTILE EXCHANGE'
MXN = 'MEXICAN PESO - CHICAGO MERCANTILE EXCHANGE'
BRL = 'BRAZILIAN REAL - CHICAGO MERCANTILE EXCHANGE'
BTC = 'BITCOIN - CHICAGO MERCANTILE EXCHANGE'
SPX = 'E-MINI S&P 500 - CHICAGO MERCANTILE EXCHANGE'

```



In case you have a request error, try applying the following code before running the import section (remember to pip install the proxy_requests library):

```

from proxy_requests.proxy_requests import ProxyRequests
req = ProxyRequests("https://api.ipify.org")
req.get()

```

Figure 11-2 shows the net COT positioning on the CAD versus the CADUSD. Notice the strong positive correlation between the two. Calculating the Pearson correlation of the last 200 observations gives a whopping 0.66. In other words, tops on the net COT data coincide with tops on the CADUSD. Similarly, troughs on the net COT data coincide with troughs on the CADUSD.

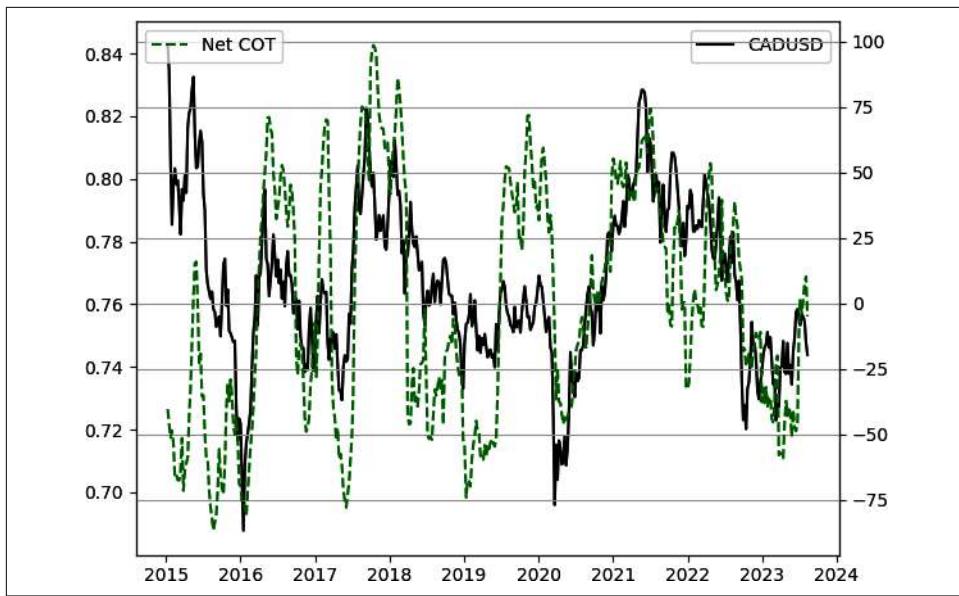


Figure 11-2. The CADUSD (left scale) versus the CAD net COT positioning (right scale).



Note that the chart shows the CADUSD and not the USDCAD, the commonly used pair. This is because you are trying to understand the CAD; therefore, it helps to use it as the base currency so that you can see the positive correlation with the CAD speculators and the negative correlation with the CAD hedgers. To obtain the CADUSD from the USDCAD observations, take its reciprocal:

$$CADUSD = \frac{1}{USDCAD}$$

The next step is to check for stationarity on the COT values so as to know whether it requires transformation or not. Remember, transformation can be either differencing, taking the percentage returns, or even using fractional differentiation (as discussed in Chapter 9):

```
from statsmodels.tsa.stattools import adfuller
print('p-value: %f' % adfuller(raw_data)[1])
```

The output is as follows:

```
p-value: 0.000717
```

The COT values seem to be stationary and ready to be used as inputs in the algorithms.

Algorithm 1: Indirect One-Step COT Model

The goodness of fit model will use long short-term memory (LSTM) to predict the next COT value at every time step. The assumption is that predicting a value that is directionally correlated to the underlying market may give a bias for the expected move during the coming week. For example, if the forecast is for a higher COT value in the coming week, then you may have a bullish CAD bias in preparation of your weekly trading.

First, import the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import import_cot_data, data_preprocessing
from master_function import plot_train_test_values,
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
```

Import the required data:

```
CAD = 'CANADIAN DOLLAR - CHICAGO MERCANTILE EXCHANGE'
data = import_cot_data(2015, 2023, CAD)
data = np.array(data.iloc[:, -1], dtype = np.float64)
```

Set the hyperparameters and create the arrays:

```
num_lags = 100
train_test_split = 0.80
num_neurons_in_hidden_layers = 200
num_epochs = 200
batch_size = 4
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data,
                                                       num_lags,
                                                       train_test_split)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Next, create the architecture of the model and predict the values on the training set (only to understand the goodness of fit) and the test set:

```
# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
```

```

model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting in the training set for illustrative purposes
y_predicted_train = model.predict(x_train)
# Predicting in the test set
y_predicted = model.predict(x_test)

```

To plot the predictions along the real values, use the following syntax:

```
plot_train_test_values(100, 50, y_train, y_test, y_predicted)
```

Figure 11-3 compares the predicted values with the real test values. At first glance, the model seems to capture the variations of the COT values well. Let's have a look at the performance results.

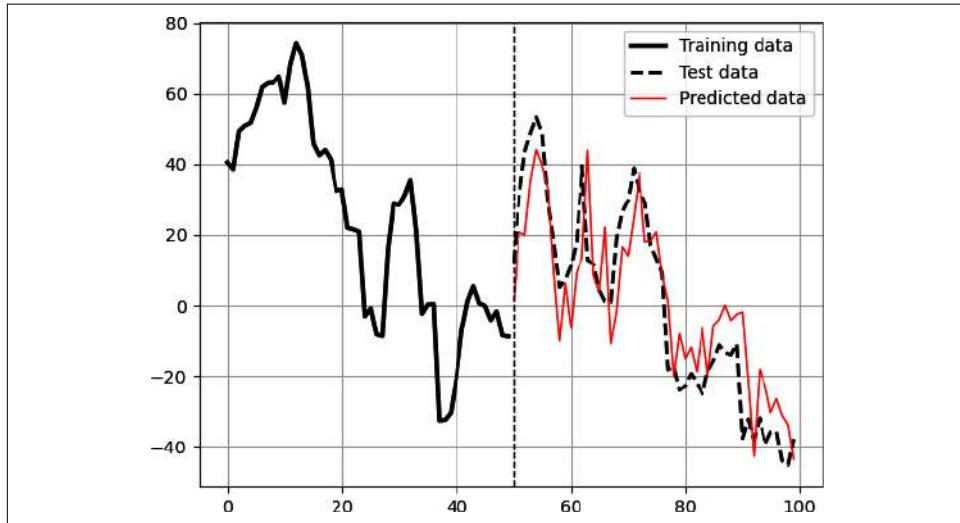


Figure 11-3. COT training data followed by COT test data (dashed line) and the predicted COT data (thin line); the vertical dashed line represents the start of the test period. The model used is the LSTM regression algorithm.

The following are the results of the model used on the CADUSD from 2015 to 2023:

```

Directional Accuracy Train = 86.18 %
Directional Accuracy Test = 60.87 %
RMSE Train = 5.3655332132
RMSE Test = 14.7772701349
Correlation In-Sample Predicted/Train = 0.995
Correlation Out-of-Sample Predicted/Test = 0.88

```



An interesting task for you would be to apply the model to forecast the returns of the underlying market using COT values as inputs. You can use either the net COT value or any of the six other series you have available, such as long hedgers and net funds. However, make sure to always check for stationarity.

Algorithm 2: MPF COT Direct Model

The MPF COT model will use LSTM to project a trajectory for the COT values to lead the way for the main market moves to come. The assumption is that by predicting the next COT values, which are less noisy than the market itself, you may have a guide for the expected trajectory of the market. As COT values are stationary and are highly correlated with the market (which is not stationary), you have a better chance of having a decent forecast than by using the MPF directly on the market. This algorithm uses the direct method (for more information, have another look at [Chapter 9](#)). First, import the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import import_cot_data, direct_mpf
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
```

Set the hyperparameters and create the arrays:

```
# Setting the hyperparameters
num_lags = 100
train_test_split = 0.80
neurons = 400
num_epochs = 200
batch_size = 10
forecast_horizon = 50
# Creating the training and test sets
x_train, y_train, x_test, y_test = direct_mpf(data,
                                               num_lags,
                                               train_test_split,
                                               forecast_horizon)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Next, create the architecture of the model and predict the values using the recursive function:

```

# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = forecast_horizon))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting in the test set
y_predicted = model.predict(x_test)

```

To plot the predictions along the real values, use the following syntax:

```

plt.plot(y_predicted[-1], label = 'Predicted data', color = 'red',
          linewidth = 1)
plt.plot(y_test[-1], label = 'Test data', color = 'black',
          linestyle = 'dashed', linewidth = 2)
plt.grid()
plt.legend()

```

Figure 11-4 compares the predicted values with the real test values.

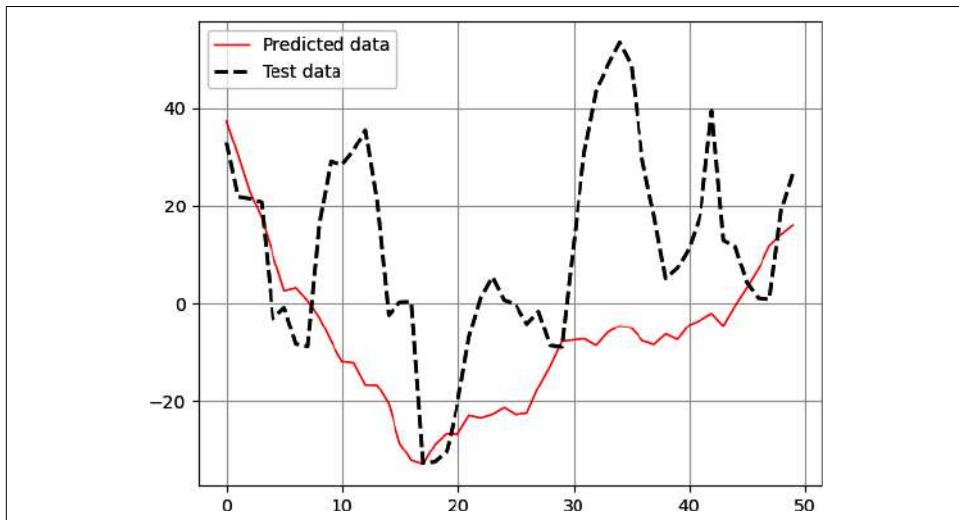


Figure 11-4. Predicted data versus test data.

The following are the results of the model used on the CADUSD:

```

Directional Accuracy Test = 57.14 %
RMSE Test = 26.4021245739
Correlation Out-of-Sample Predicted/Test = 0.426

```

Algorithm 3: MPF COT Recursive Model

This algorithm uses the recursive method (for more information, have another look at [Chapter 9](#)). First, import the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import data_preprocessing, import_cot_data
from master_function import plot_train_test_values, recursive_mpf
from master_function import calculate_directional_accuracy
```

Set the hyperparameters and create the arrays:

```
num_lags = 100
train_test_split = 0.80
neurons = 100
num_epochs = 200
batch_size = 2
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data,
                                                       num_lags,
                                                       train_test_split)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Next, create the architecture of the model and predict the values using the recursive function:

```
# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
# Fitting the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting in the test set on a recursive basis
x_test, y_predicted = recursive_mpf(x_test, y_test, num_lags, model)
```

To plot the predictions along the real values, use the following syntax:

```
plot_train_test_values(100, 50, y_train, y_test, y_predicted)
```

Figure 11-5 compares the predicted values with the test values.

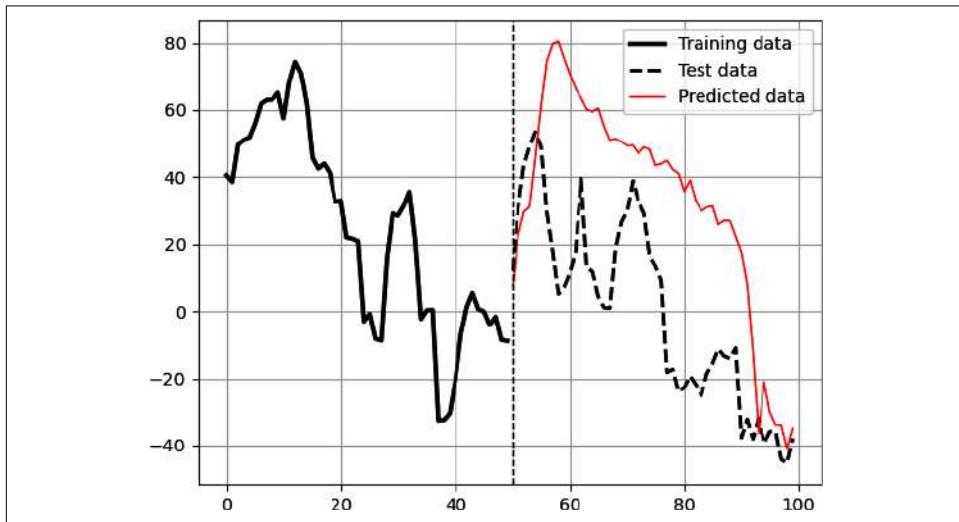


Figure 11-5. Multiperiod forecasting of the COT data.

The following are the results of the model used on the CADUSD from 2015 to 2023:

Directional Accuracy Test = 52.17 %
RMSE Test = 40.3120541504
Correlation Out-of-Sample Predicted/Test = 0.737



Remember that while setting the `random_state` and ensuring reproducibility is useful for experimentation, it might also limit the model's ability to generalize well if the data distribution is genuinely random. In many cases, it's recommended to perform multiple runs with different random seeds to get a better sense of the model's performance and robustness. This is why many of the models seen in this book do not have the `random_state` implementation in their code (except a few models in Chapter 7).

It does not hurt to remember the symptoms of overfitting. Unfortunately, overfitting is sometimes not that easily detectable, so consider these general rules:

High training accuracy and low test accuracy

The model shows excellent performance on the training data but performs poorly on the test data. This is a clear indication that the model has memorized the training data rather than learning general patterns.

Large gap between training and test performance

There is a significant difference between the training and test error rates. Ideally, the two measures should be close to each other, and a large gap suggests overfitting.

Unusually high model complexity

If the model is overly complex with a large number of parameters or features, it becomes more prone to overfitting. A simpler model may generalize better to new data.

Noisy predictions

Overfit models tend to make erratic and inconsistent predictions on new data. This is because they are highly sensitive to small variations in the input data, including noise.

Putting It All Together

The COT report, released every Friday by the CFTC, outlines the positioning of key market participants. It can be transformed into a time series forecasting task with the aim of improving market forecasts. The market participants of particular interest are the commercial participants (dealers or hedgers) and the noncommercial participants (leveraged money or funds).



Make sure to put *master_function.py* in the directory of the interpreter. Alternatively, you can simply open *master_function.py* in Python and execute the whole file. However, the latter method requires you to do it every time you restart the kernel.

Deep learning techniques can be applied on COT values to forecast market positioning through hidden patterns and seasonal configurations. This section discussed three algorithms to squeeze out value from the COT report. You can experiment with machine learning models, deep reinforcement learning, and even simple statistical techniques to better understand the COT data.

To manually import COT values, refer to the historical data found in the [GitHub repository](#) and use the following code (change the path of the interpreter to be the same as the location of the downloaded file):

```
import pandas as pd
import numpy as np
# Import the data using pandas
data = pd.read_excel('name_of_file.xlsx')
```

It's important to remember that the COT report provides a snapshot of market participant positioning and should be used in conjunction with other tools and analysis methods. While it can offer valuable information, it's not a standalone trading strategy, and careful consideration of other factors is essential for making well-informed trading decisions.

Using Technical Indicators as Inputs

You learned about technical indicators in [Chapter 5](#). It's time to use them as inputs to predict the underlying market's returns. Using lagged values implies that there must be value in the past observations, which may translate into decent forecasts. This section will explore past that assumption and will search for value in other price transformations. You can have the following price-derived calculations:

Mathematical transformation

This type of transformation is likely to be a direct manipulation of the raw data. An example of this is a basic differencing or a simple moving average. Normalization is also part of mathematical transformation.

Technical transformation

This type of transformation is less obvious, and the result may not look at all like the raw data. An example of this is the RSI, a technical indicator created out of recursive rules based on moving averages and normalization.

Categorical transformation

This type of transformation shifts the type of numerical data to categorical and vice versa. For example, an algorithm that is known as *OneHotEncoder* takes categorical data and transforms it into binary data so that machine learning algorithms are able to classify it.

Before proceeding, there is a data issue that is worth discussing. *Multicollinearity* is a statistical phenomenon that occurs in regression analysis when two or more independent variables (inputs) in a multiple regression model are highly correlated with each other. In other words, it is a situation where there is a strong linear relationship between two or more of the predictors. This correlation can make it difficult for the regression model to separate the individual effects of each predictor on the dependent variable (the outcome variable). Obviously, if you are using multiple RSIs with different time periods, then you are likely to run into multicollinearity. Make sure you look for weakly correlated indicators.

Two technical indicators (or transformations) are used in this example:

- The five-week RSI, a stationary indicator that does not require any transformation.
- The difference between the weekly close price and the 20-week moving average. This is also a stationary calculation that does not require any transformation.

Therefore, the EURUSD's weekly returns will be forecasted using the previous week's RSI value and the distance between the previous week's close price and the 20-week moving average.

First, import the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import mass_import, rsi, ma, calculate_accuracy
from master_function import plot_train_test_values,
from master_function import multiple_data_preprocessing
from sklearn.metrics import mean_squared_error
from master_function import add_column, delete_column
```

Next, import the data using the `mass_import()` function:

```
data = mass_import(0, 'W1')[ :, -1]
data = rsi(np.reshape(data, (-1, 1)), 5, 0, 1)
data = ma(data, 5, 0, 2)
data[:, 2] = data[:, 0] - data[:, 2]
data = add_column(data, 1)
for i in range(len(data)):
    data[i, 3] = data[i, 0] - data[i - 1, 0]
data[:, 0] = data[:, -1]
data = delete_column(data, 3, 1)
```

Define the multiple data preprocessing function, which takes the values of the two technical indicators and lags them so that they can be used as inputs to predict the EURUSD's returns. **Table 11-2** shows the training table `x_train` with six lagged values as independent variables to explain and predict the next EURUSD return.

Table 11-2. A sample of the training array

RSI t-1	(Close – MA)t-1	RSI t-2	(Close – MA)t-2	RSI t-3	(Close – MA)t-3
36.6190	-0.003804	48.5188	0.001044	42.4396	-0.001714
46.7928	0.001674	36.6190	-0.003804	48.5188	0.001044
40.5430	-0.002518	46.7928	0.001674	36.6190	-0.003804
65.9614	0.011340	40.5430	-0.002518	46.7928	0.001674
47.2585	-0.000390	65.9614	0.011340	40.5430	-0.002518
63.9755	0.011302	47.2585	-0.000390	65.9614	0.011340

The next function is already defined, but it doesn't hurt to discuss what it does. The `multiple_data_preprocessing()` function simply creates the four needed arrays for the backtest, but it uses the two technical indicators as inputs. You can define the function for six lagged values (three for the RSI and three for the difference between the close and the moving average) as follows:

```
def multiple_data_preprocessing(data, train_test_split):
    data = add_column(data, 4)
    data[:, 1] = np.roll(data[:, 1], 1, axis = 0)
    data[:, 2] = np.roll(data[:, 2], 1, axis = 0)
    data[:, 3] = np.roll(data[:, 3], 1, axis = 0)
    data[:, 4] = np.roll(data[:, 4], 1, axis = 0)
    data[:, 5] = np.roll(data[:, 5], 1, axis = 0)
    data[:, 6] = np.roll(data[:, 6], 1, axis = 0)
    data = data[1:, :]
    x = data[:, 1:]
    y = data[:, 0]
    split_index = int(train_test_split * len(x))
    x_train = x[:split_index]
    y_train = y[:split_index]
    x_test = x[split_index:]
    y_test = y[split_index:]
    return x_train, y_train, x_test, y_test
```

Set the hyperparameters and create the arrays:

```
num_lags = 6 # Must equal the number of the lagged values you create
train_test_split = 0.80
neurons = 500
num_epochs = 500
batch_size = 200
# Creating the training and test sets
x_train, y_train, x_test, y_test = multiple_data_preprocessing(data,
                                                               train_test_split)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Next, create the architecture of the model and predict the values on the training set (only to understand the goodness of fit) and the test set:

```
# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a third layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a fourth layer
```

```

model.add(Dense(neurons, activation = 'relu'))
# Adding a fifth layer
model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'ada')
# Fitting the model
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
# Predicting in the training set for illustrative purposes
y_predicted_train = model.predict(x_train)
# Predicting in the test set
y_predicted = model.predict(x_test)

```

To plot the predictions along the real values, use the following syntax:

```
plot_train_test_values(100, 50, y_train, y_test, y_predicted)
```

Figure 11-6 compares the predicted values with the real test values.

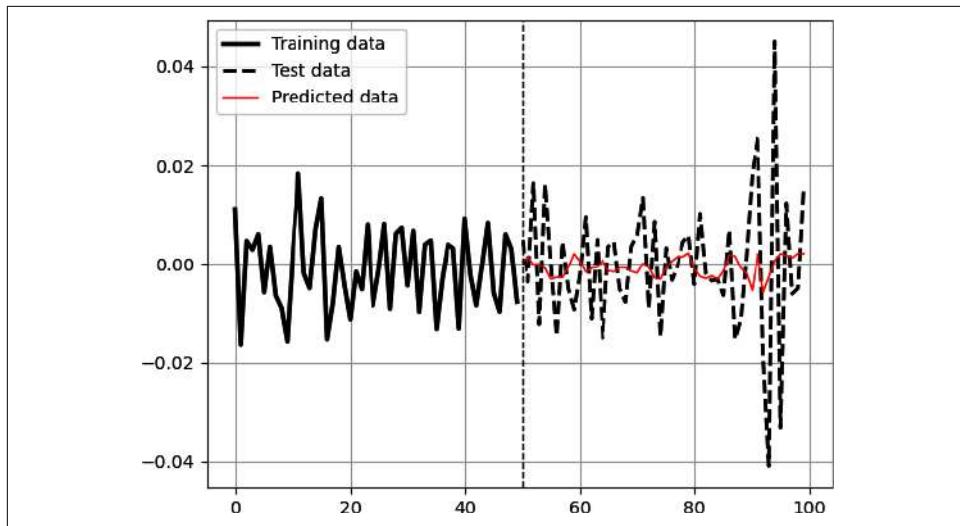


Figure 11-6. Training data followed by test data (dashed line) and the predicted data (thin line); the vertical dashed line represents the start of the test period. The model used is the LSTM regression algorithm.

The following are the results of the model:

```

Accuracy Train = 59.39 %
Accuracy Test = 51.82 %
RMSE Train = 0.0163232045
RMSE Test = 0.0122093494
Correlation In-Sample Predicted/Train = 0.255
Correlation Out-of-Sample Predicted/Test = 0.015

```

It is very interesting to tweak the model and see how to improve it. As these are weekly predictions, good accuracy could be the first step in shaping your swing trading on the condition of optimizing the model and making sure it is not overfit.¹ It is worth noting that since this algorithm is trying to predict the financial returns of an instrument, the `calculate_accuracy()` function is used as opposed to `calculate_directional_accuracy()`.



Try running the algorithm seen in this section on MPF mode and see what you can extract from it. Remember the algorithm's limitations while doing so.

Predicting Bitcoin's Volatility Using Deep Learning

Bitcoin is a decentralized digital currency that was created in 2009 by an unknown person or entity using the pseudonym *Satoshi Nakamoto*. It was the first cryptocurrency to be introduced and remains the most well known and widely traded cryptocurrency to date. You probably do not need an introduction to Bitcoin considering the immense hype over it during these past years, but more knowledge never hurts.

Bitcoin operates on a technology called *blockchain*, which is a distributed ledger system. Unlike traditional currencies that are issued and regulated by governments or central banks, Bitcoin is not controlled by any central authority.

Instead, it relies on a peer-to-peer network of computers, known as nodes, to validate and record transactions. Nowadays, Bitcoin is heavily traded on cryptocurrency exchanges and is used for speculative but also hedging operations. The most commonly traded pair is BTCUSD, which is the value of 1 bitcoin relative to USD.

Figure 11-7 shows the evolution of the BTCUSD (Bitcoin's value priced in US dollars).

¹ Swing trading involves holding positions for a short- to medium-term period, typically a few days to a few weeks, to profit from price swings or price movements within a larger trend.

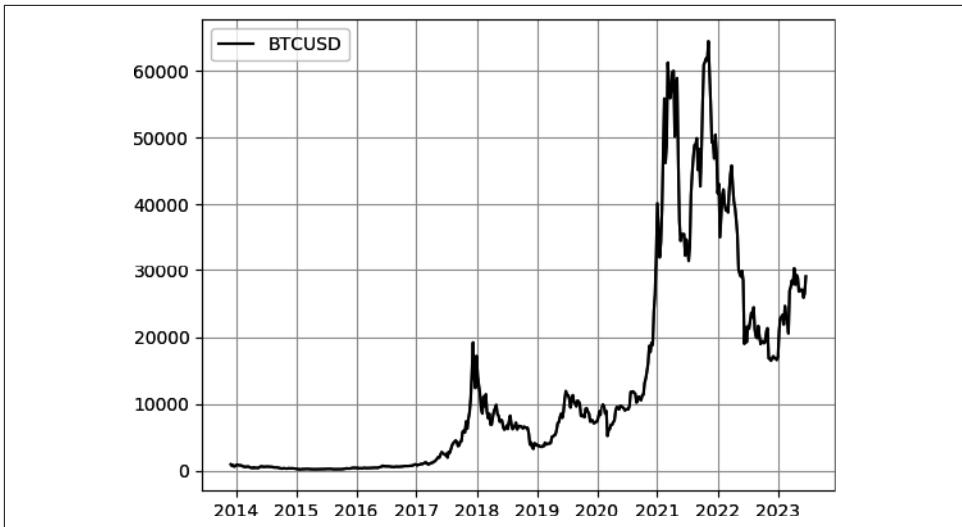


Figure 11-7. Historical evolution of BTCUSD since 2014 in a linear scale.

You can generate [Figure 11-7](#) using the following code:

```
import pandas as pd
# Manually importing BTCUSD values
my_data = pd.read_excel('Bitcoin_Daily_Historical_Data.xlsx')
# Renaming the columns
my_data.columns = ['Date', 'Open', 'High', 'Low', 'Close']
# Setting the date column
my_data['Date'] = pd.to_datetime(my_data['Date'])
# Charting
plt.plot(my_data['Date'], my_data['Close'], label = 'BTCUSD',
         color = 'black')
plt.legend()
plt.grid()
```

Make sure you download the historical BTCUSD values from the [GitHub repository](#) and that you set the directory of the interpreter in the same folder as the downloaded file.

[Figure 11-7](#) is charted using a linear scale, which means that data is represented on a straight and evenly spaced axis (the space between 10 and 20 is the same as the space between 1230 and 1240). It is also possible to use what is known as a logarithmic scale to chart time series that experience big jumps.

To Log or to Lin

The *logarithmic scale* is a scale used in charting that represents the data using logarithmic transformations. In a logarithmic scale, the spacing between values is based on the logarithm of the actual values rather than their linear scale. This means that the distance between, for example, 1 and 10 on a logarithmic scale is the same as the distance between 10 and 100, rather than being 10 times larger as in a linear scale.

The importance of using a logarithmic scale in charting financial time series stems from the nature of financial data and its tendency to exhibit exponential growth or decay (as is the case on BTCUSD).

Financial markets often experience large variations in prices or returns, and using a logarithmic scale can help to better visualize and understand these changes. In trading jargon, linear scale charts are typically referred to as *lin charts*, and logarithmic scale charts are referred to as *log charts*.

Using a logarithmic scale in charting financial time series helps provide a more accurate representation of the data, facilitates the analysis of growth rates, and enhances the ability to identify trends and patterns in financial markets.

Figure 11-8 shows the evolution of BTCUSD (Bitcoin's value priced in US dollars) in logarithmic scale.

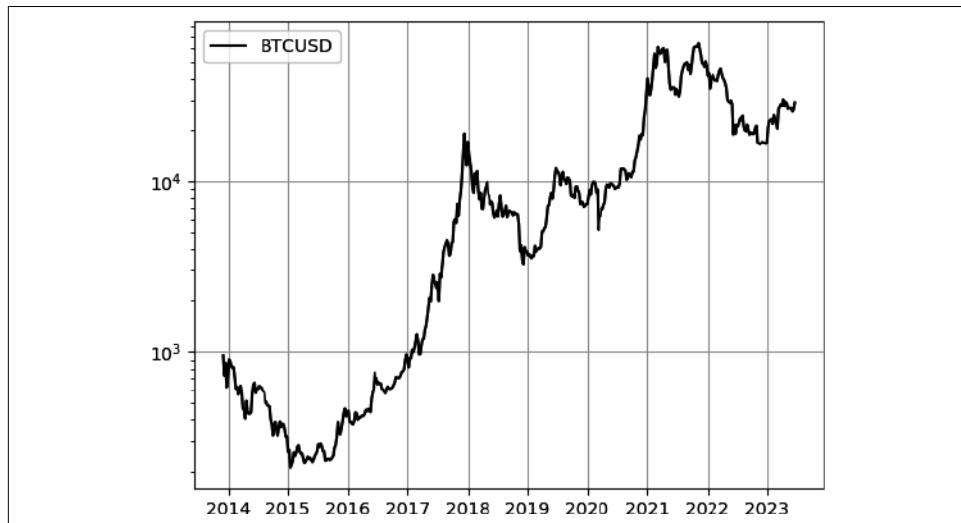


Figure 11-8. Historical evolution of BTCUSD since 2014 in a semilog scale.

Use the following code to generate **Figure 11-8** using the `plt.semilogy()` function:

```
plt.semilogy(my_data['Date'], my_data['Close'], label = 'BTCUSD',
             color = 'black')
plt.legend()
plt.grid()
```



The word *semilog* refers to transforming one of the two axes into a logarithmic scale, while the word *log* refers to transforming both axes into a logarithmic scale.

Since time is linear, you only need to transform the *y*-axis (which means the values), and therefore, you are technically using semilog charts. This is why the `matplotlib()` function is called `plt.semilogy()`.

If you prefer to import BTCUSD values using a Python script, use the following:

```
import requests
import json
import pandas as pd
import numpy as np
import datetime as dt

# Selecting the time frame
frequency = '1h'
# Defining the import function
def import_crypto(symbol, interval = frequency):
    # Getting the original link from Binance
    url = 'https://api.binance.com/api/v1/klines'
    # Linking the link with the cryptocurrency and the time frame
    link = url + '?symbol=' + symbol + '&interval=' + interval
    # Requesting the data in the form of text
    data = json.loads(requests.get(link).text)
    # Converting the text data to a dataframe
    data = np.array(data)
    data = data.astype(np.float)
    data = data[:, 1:5]
    return data
# Importing hourly BTCUSD data
data = import_crypto('BTCUSDT')
```

Let's see if deep learning helps forecast BTCUSD's volatility using its lagged values. But first, two questions are begging an answer:

- What is the use of predicting Bitcoin's volatility?
- How do you calculate Bitcoin's volatility?

To answer the first question, by predicting Bitcoin's volatility, traders can potentially identify periods of increased price swings and capitalize on them. Similarly, volatility predictions can also provide insights into market sentiment. When investors expect uncertain or turbulent market conditions, it may indicate a lack of confidence, leading to potential changes in market dynamics.

To answer the second question, you can use a rolling standard deviation measure on the close prices. This allows you to create a new time series that reflects the historical volatility of Bitcoin.



Typically, an increase in volatility is a sign of market stress and fear, which translates to a bearish tone. In contrast, a decrease in volatility is a sign of a healthy and stable market, which translates to a bullish market. This relationship is not perfect, and other variables may impact it. For example, if you calculate the correlation coefficient of Bitcoin's daily close prices and the 10-day rolling volatility, you will find that it's positive (at some periods, it's extremely positive). Bitcoin is known for being a euphoric asset where rises are accompanied by a phenomenon called FOMO, an abbreviation of *fear of missing out*. This psychological bias is one of the building blocks of a market bubble where everyone keeps buying in the hopes of profiting from the move.

The first step of predictive analytics is to understand the data you're dealing with. As a reminder, volatility refers to the degree of variation or fluctuation in the price of a financial instrument, such as a stock, bond, commodity, or currency, over a specific period of time. It is a statistical measure of the dispersion for that particular asset.

Figure 11-9 shows the latest values of Bitcoin's 10-day volatility as measured by the rolling standard deviation calculation. The latest values say that recently, the price variations hovered around \$500 from the 10-day mean most of the time (refer to [Chapter 3](#) for more in-depth comprehension on standard deviation).

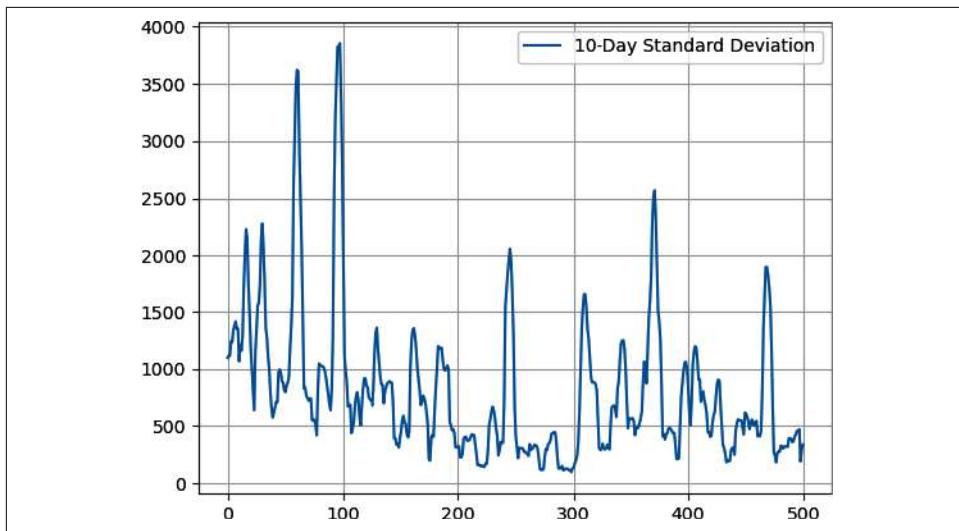


Figure 11-9. Bitcoin's rolling 10-day standard deviation as a proxy for volatility.

Let's get started. For this task, let's manually import BTCUSD into the interpreter. First, start by importing the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import add_column, delete_row, volatility
from master_function import data_preprocessing, plot_train_test_values
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
from statsmodels.tsa.stattools import adfuller
```

Next, import the data using *pandas*:

```
data = pd.read_excel('Bitcoin_Daily_Historical_Data.xlsx').values
```

The next step is to calculate volatility. Its function is defined in *master_function*, and as you have imported it already, it should work directly:

```
data = volatility(data, 10, 0, 1)
data = data[:, -1]
```

You have to check for stationarity. If the data is stationary, then it's ready to be deployed for training. Otherwise, you may have to transform it. The following code applies the ADF test on the 10-day volatility of Bitcoin:

```
print('p-value: %f' % adfuller(data)[1])
```

Here is the output:

```
p-value: 0.120516
```

It seems that the volatility values are nonstationary. Let's try differencing:

```
data = np.diff(data)
```



You can also apply fractional differentiation to preserve a hint of memory.

Figure 11-10 shows the latest values of Bitcoin's 10-day differenced volatility.

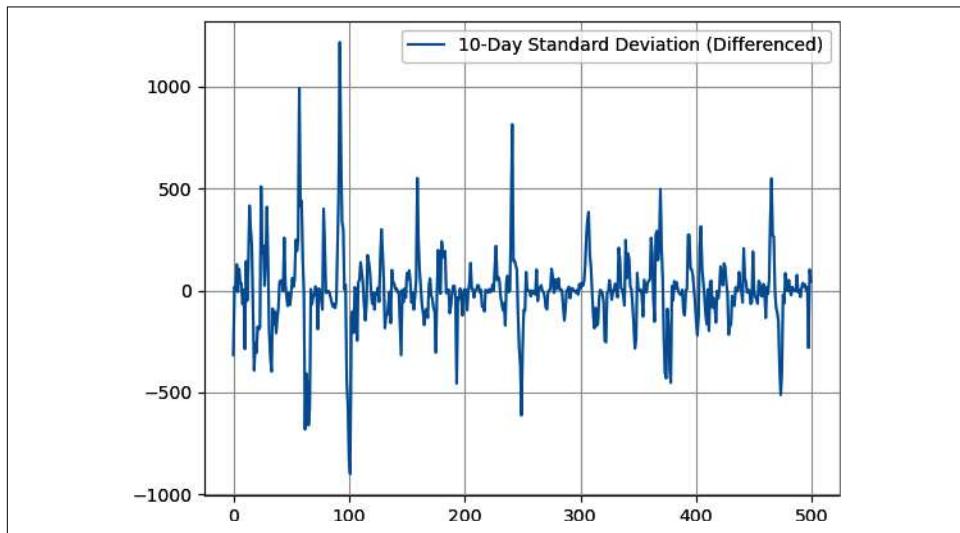


Figure 11-10. Bitcoin's rolling 10-day standard deviation (differenced).

The next step is to set the hyperparameters and prepare the arrays as usual:

```
num_lags = 300
train_test_split = 0.80
neurons = 80
num_epochs = 100
batch_size = 500
# Prepare the arrays
x_train, y_train, x_test, y_test = direct_mpf(data,
                                                num_lags,
                                                train_test_split,
                                                forecast_horizon)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Create the deep neural network architecture with a few extra layers:

```
# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a third layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a fourth layer
model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = forecast_horizon))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
```

Now, fit and predict the data:

```
model.fit(x_train, y_train, epochs = num_epochs, batch_size = batch_size)
y_predicted = model.predict(x_test)
```

Figure 11-11 compares the predicted values with the test values.

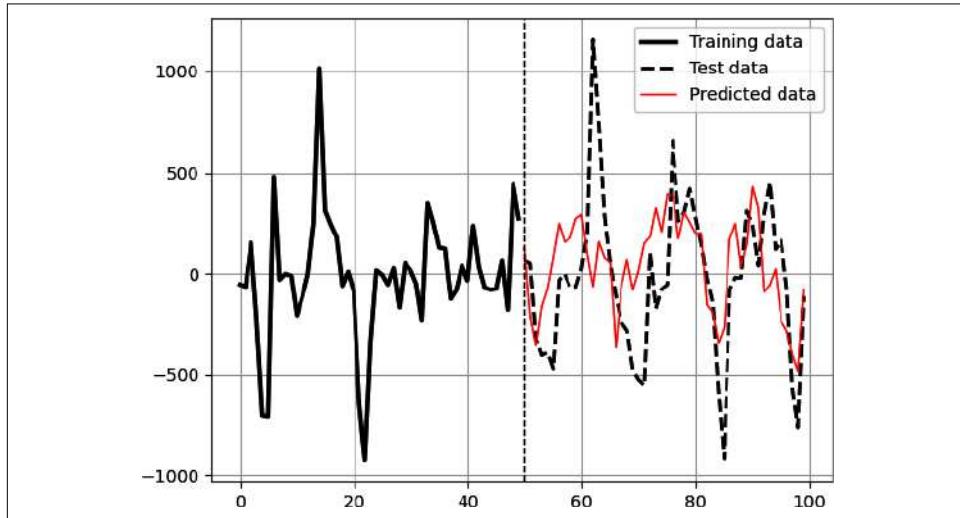


Figure 11-11. Forecasting Bitcoin's volatility; the model used is the LSTM regression algorithm.

The following are the results of the model:

```
Accuracy Train = 66.56 %
Accuracy Test = 63.59 %
RMSE Train = 95.6086778521
RMSE Test = 186.1401365824
Correlation In-Sample Predicted/Train = 0.807
Correlation Out-of-Sample Predicted/Test = 0.56
```

Keep in mind that the backtesting results may differ significantly due to the backtested period, time granularity, transaction costs, different quotations from different brokers, different hyperparameters, and different randomization. Optimization is a key component, and validation of the results must be done before forming an opinion on the algorithm. Your task is therefore to improve the results and get a better prediction on volatility.

You may also wonder if you can predict Bitcoin's returns directly with an accuracy better than random. The answer is yes, and if you need a few ideas of inputs that may help predict Bitcoin's returns, check out the following nonexhaustive list:

- *Historical price data and its derivatives*: Using historical price data is a fundamental aspect of predicting returns. You can include features such as daily, weekly, or monthly price changes, moving averages, and price volatility.
- *Trading volume*: The trading volume of Bitcoin provides valuable information about the level of market activity and liquidity. Higher trading volumes often accompany significant price movements.
- *Market sentiment indicators*: Sentiment analysis from social media platforms, news articles, or forums can help gauge the overall market sentiment toward Bitcoin. Bitcoin's Fear & Greed index is a good candidate as it's published on a daily basis and uses many fundamental variables to calculate its values.²
- *Network metrics*: Bitcoin's blockchain data provides useful metrics, such as the number of transactions, hash rate, and difficulty level. These metrics reflect the activity and health of the Bitcoin network, which can influence prices.
- *Market indicators*: Consider using general market indicators like the S&P 500 or the VIX as external variables. Cryptocurrencies, including Bitcoin, can sometimes exhibit correlations with traditional financial markets.

² The index is used to gauge the emotions and sentiments of investors in the cryptocurrency market. It provides a numerical value on a scale from 0 to 100, where lower values indicate extreme fear and higher values indicate extreme greed. The index is designed to help traders and investors identify potential market turning points based on prevailing emotions.

- *Cryptocurrency-specific indicators*: Other indicators specific to the cryptocurrency space, such as the total market capitalization of all cryptocurrencies and dominance ratio of Bitcoin, may provide insights into broader market conditions.
- *Technical indicators*: Various technical analysis indicators, such as the RSI and volatility, can offer insights into potential price trends and reversals.
- *Google trends*: Monitoring the popularity of search terms related to Bitcoin on Google Trends can provide insights into public interest and potential price movements.
- *Cryptocurrency exchange data*: Data from cryptocurrency exchanges, such as open interest, funding rates, and liquidation data, can offer insights into market dynamics and potential price shifts.

Real-Time Visualization of Training

What happens during training? Sure, you can see that the training process is ongoing when you look at the progress bar of every epoch:

Epoch 1/100

9/9 [=====] – 3s 77ms/step - loss: 0.0052

Epoch 2/100

9/9 [=====] – 1s 78ms/step - loss: 0.0026

Epoch 3/100

9/9 [=====] – 1s 68ms/step - loss: 0.0015

However, you can also code a dynamic plot that shows you the in-sample predictions getting updated through epochs as they approach the in-sample real values. This will be the first aim of this section. Before proceeding, refresh your knowledge of the terminologies:

In-sample real values

These are the values contained in `y_train`. They are the real values the model uses to calibrate its training. They belong to the training set.

In-sample predictions

These are the values contained in `y_predicted_train`. They are the predictions the model outputs during its training. They belong to the training set and suffer from *look-ahead bias*, which happens when future information is used to make decisions or predictions that should have been made in the past based only on historical data available at that time.

Out-of-sample real values

These are the test values used to test the model's ability to predict values on data that has never been seen before. They belong to the test set.

Out-of-sample predictions

These are the predictions that follow the training phase. They belong to the test set.

Let's take an example that you are familiar with from [Chapter 9](#), the ISM PMI data. The aim is to create a one-step forecast LSTM model of the differenced ISM PMI values while creating a dynamic plot during training that shows predictions being calibrated to the training set. First, import the required libraries:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, LSTM
from master_function import data_preprocessing
from master_function import calculate_directional_accuracy
from sklearn.metrics import mean_squared_error
import tensorflow as tf
import random
```

From the [GitHub repository](#), import and difference the ISM PMI data:

```
data = np.diff(np.reshape(pd.read_excel('ISM_PMI.xlsx').values, (-1)))
```

The next step is to set the hyperparameters and prepare the arrays as usual:

```
num_lags = 100
train_test_split = 0.80
neurons = 200
num_epochs = 200
batch_size = 4
# Creating the training and test sets
x_train, y_train, x_test, y_test = data_preprocessing(data,
                                                       num_lags,
                                                       train_test_split)
```

To comply with the LSTM architecture, the independent variables must be transformed into three-dimensional arrays. This is done using the following code:

```
x_train = x_train.reshape((-1, num_lags, 1))
x_test = x_test.reshape((-1, num_lags, 1))
```

Create the deep neural network architecture:

```
# Create the LSTM model
model = Sequential()
# Adding a first layer
model.add(LSTM(units = neurons, input_shape = (num_lags, 1)))
# Adding a second layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a third layer
model.add(Dense(neurons, activation = 'relu'))
# Adding a fourth layer
model.add(Dense(neurons, activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1))
# Compiling the model
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
```

The following code fits the `x_train` data to the `y_train` data while plotting the predictions at every epoch:

```
def update_plot(epoch, logs):
    if epoch % 1 == 0:
        plt.cla()
        y_predicted_train = model.predict(x_train)
        plt.plot(y_train, label = 'Training data',
                  color = 'black', linewidth = 2.5)
        plt.plot(y_predicted_train, label = 'Predicted data',
                  color = 'red', linewidth = 1)
        plt.title(f'Training Epoch: {epoch}')
        plt.xlabel('Time')
        plt.ylabel('Value')
        plt.legend()
        plt.savefig(str(random.randint(1, 1000)))
# Create the dynamic plot
fig = plt.figure()
# Train the model using the on_epoch_end callback
class PlotCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs = None):
        update_plot(epoch, logs)
        plt.pause(0.001)
plot_callback = PlotCallback()
history = model.fit(x_train, y_train, epochs = num_epochs,
                     batch_size = batch_size, callbacks = [plot_callback])
```

Figure 11-12 shows the training at epoch 1. Notice how the algorithm is just starting out and is not quite capturing the relationship between the independent and dependent variables yet.

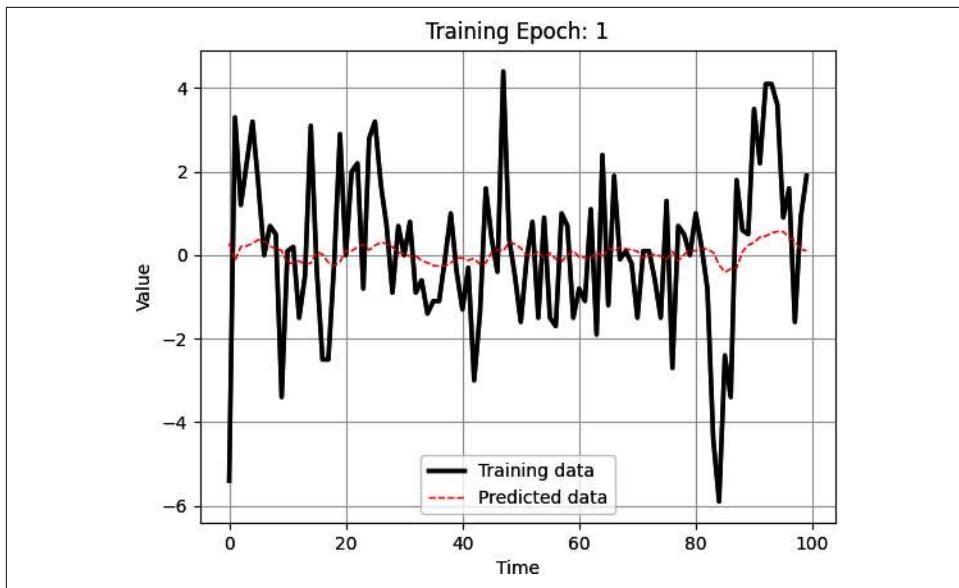


Figure 11-12. ISM PMI data training in progress at epoch 1.

Figure 11-13 shows the training at epoch 21. It looks like the algorithm is still calibrating itself to the features.

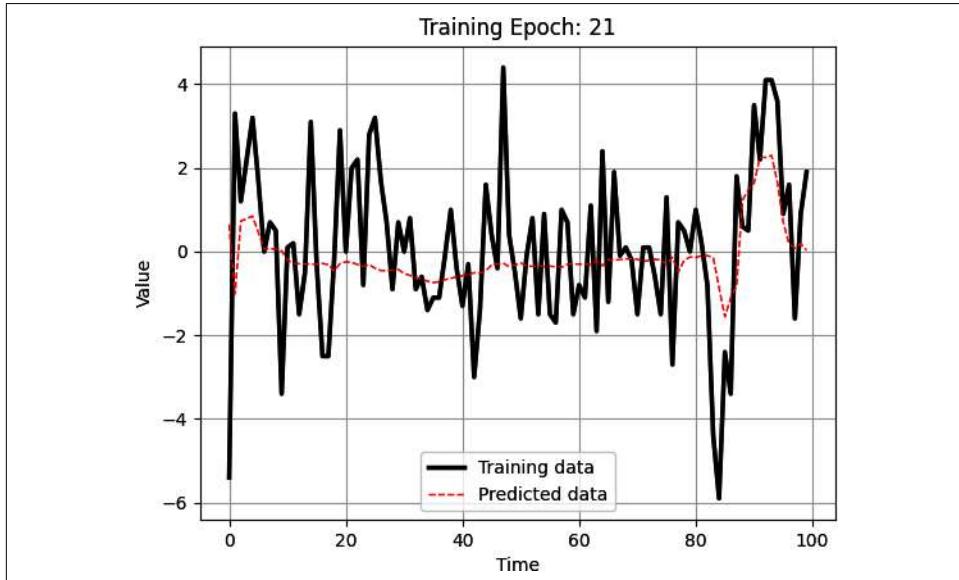


Figure 11-13. ISM PMI data training in progress at epoch 21.

Figure 11-14 shows the training at epoch 29. The model is starting to properly fit the data.

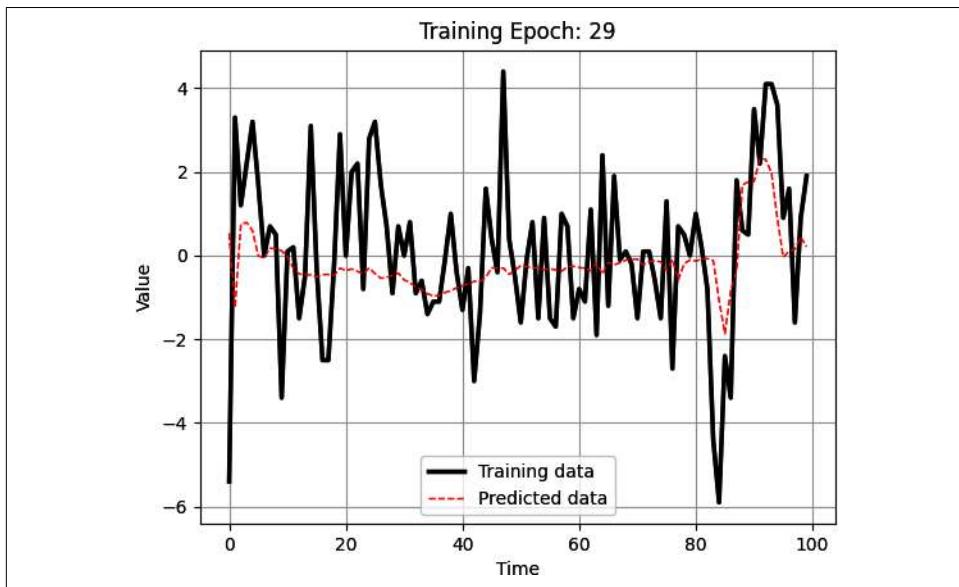


Figure 11-14. ISM PMI data training in progress at epoch 29.

Figure 11-15 shows the training at epoch 62. The model seems to fit the data well, although with some errors, but this is not the main aim of this section.

Dynamic training plots can be an interesting tool to see how the model is learning—and whether it's actually learning something. This helps with the problem of *constant predictions*, which occur when the model fails to capture any relationship between the dependent and independent variables.

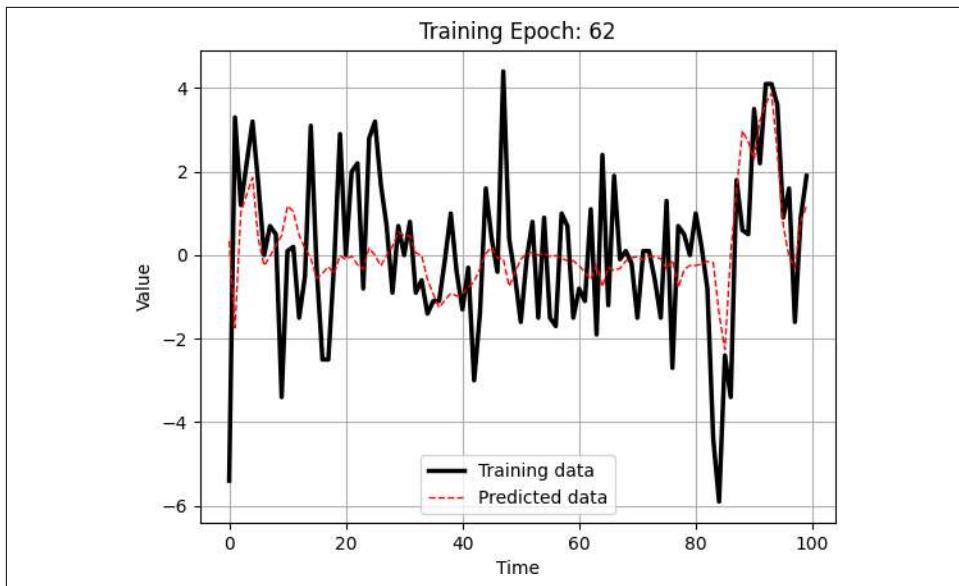


Figure 11-15. ISM PMI data training in progress at epoch 62.

Several factors can lead to the problem of constant predictions in deep learning:

Poor model architecture

If the model is not expressive enough, it may struggle to learn meaningful patterns, resorting to a simple, constant prediction instead.

Limited or noisy data

Insufficient or noisy data can hinder the model's ability to learn meaningful patterns. If the data lacks diversity or contains significant errors, the model may converge to a constant forecast as the simplest way to minimize the loss.

Improper loss function

The choice of the loss function plays a vital role in guiding the model during training. If the loss function is not appropriate for the task or the model architecture, it may not encourage the model to make varied predictions, leading to constant forecasts.

Poor hyperparameter tuning

Hyperparameters, such as the batch size and the number of neurons, can significantly impact the training process. If these hyperparameters are not appropriately tuned, the model might not converge effectively, leading to constant predictions.

At this point, you must have wondered what the architecture you built using the `Sequential()` and `Dense()` functions looks like. Naturally, it should look like the neural network graphs you have seen previously. To see this, you must `pip install` the required libraries from the prompt:

```
pip install pydot
pip install pydotplus
pip install graphviz
```

Then download the `graphviz` binaries folder from the [official website](#), extract the contents of the file, then set the `bin` folder as one of the paths of the Python interpreter (e.g., Spyder). Restart the kernel, and proceed to compile your model as usual. Finally, use the following code to print the architecture:

```
from tensorflow.keras.utils import plot_model
from IPython.display import Image
plot_model(model, to_file = 'Architecture.png',
            show_shapes = True,
            show_layer_names = True)
Image('Architecture.png')
```

Figure 11-16 shows the output of the code.

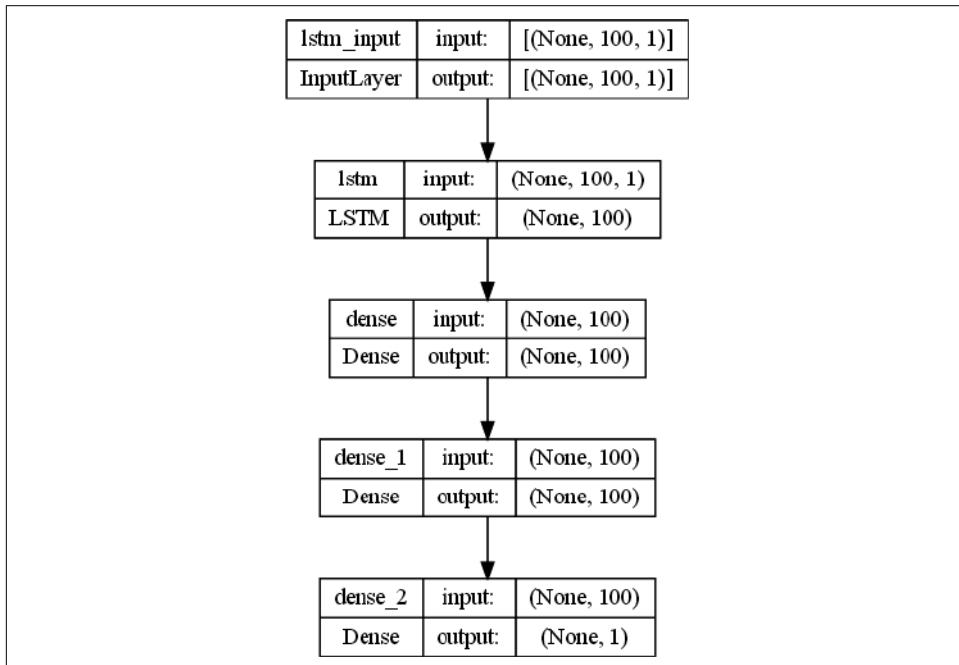


Figure 11-16. Model architecture example.

The code outputs the current architecture of the compiled LSTM model.

Summary

This information-heavy chapter showed you a few ideas for using a selection of trading algorithms to forecast returns. It is mostly a way to stimulate critical and innovative thinking and to find new and innovative ideas from where trading signals may be derived. For example, you can try applying filters with the signals you get from the algorithms. *Filters* are like on/off switches that allow the signal based on whether a final condition is met or not. An example of a hypothetical trading strategy with a filter is to take the bullish signals only if the market is above its 200-day moving average and to take the bearish signals only if the market is below its 200-day moving average.

Your main takeaway should be how things must be structured so that you can understand the backtesting process. Compared to this chapter, [Chapter 12](#) will be a gentle breeze as it explores risk management and fundamental tools aimed at enhancing the research process.

Market Drivers and Risk Management

In today's rapidly evolving markets, understanding market drivers and risk management is essential for achieving success in trading and investing endeavors.

Market drivers are the factors that influence financial markets, such as economic indicators, corporate performance, geopolitical events, central bank policies, and technological advancements. By understanding these drivers, investors can make informed decisions and anticipate market trends. Algorithmic traders must be aware of these events that may cause turbulence in their models. *Risk management* is the backbone of a solid investment strategy. It encompasses diversification, stop-loss orders, position sizing, risk-reward assessment, and emotional discipline.

By combining a deep understanding of market drivers with effective risk management techniques, traders can navigate the complexities of financial markets and increase the likelihood of a successful trading algorithm.

This chapter is divided into two main sections. The first section talks about market drivers in a fundamental sense, and the second section discusses risk management from a general point of view.

Market Drivers

Knowledge of market drivers is essential to develop a healthy and sound trading system. Failing to understand what pushes the markets will inevitably cause the demise of the whole process. Up to this point you have seen quantitative trading algorithms that use either lagged values of the same time series or technical indicators, but that's only a one-dimensional way of seeing things.

During trading, many events occur that can impact the positions and the risk taken. Knowledge of these events may help mitigate some of the risk, but it may also offer a few directional opportunities along the way. This section can be considered a primer on fundamental analysis and market expectations. If you are familiar with this field, then it should be a breeze.

Market Drivers and Economic Intuition

Market drivers influence and lead the movement and behavior of assets. These drivers can be financial, economic, geopolitical, or even psychological. Consider the following four asset classes: stocks, fixed income, commodities, and currencies.



As a reminder, *fixed income* refers to a type of investment that provides investors with regular and predictable payments in the form of interest or dividends over a set period. It is called fixed income because the income generated from these investments is typically fixed and known in advance, as opposed to variable income generated by other investments like stocks.

The most common form of fixed income investments is *bonds*, which are essentially debt securities issued by governments, municipalities, corporations, or other entities to raise capital.

When the economy is strong, it generally has different effects on asset classes due to the varying preferences and behaviors of investors and market participants. Let's look at them in detail:

Stock market

Strong economic conditions often lead to higher corporate profits as businesses experience higher revenues and sales. This boost in earnings can drive stock prices higher, as investors expect better returns from companies. It also results in higher consumer spending. This can benefit companies that rely heavily on consumer purchases, such as retail and consumer goods companies, which may experience increased sales and profitability. Additionally, a strong economy instills confidence in investors, leading them to be more willing to take on risk and invest in the stock market (as opposed to the fixed income market). Positive sentiment can attract more buyers, leading to a rise in stock prices (which may also attract speculative traders who will push prices even higher). It is also possible that central banks may implement policies to effectively control inflation, which can be positive for stocks. Moderate inflation rates and low interest rates can encourage borrowing, investment, and business expansion.

Fixed income market

As the economy strengthens, central banks may increase interest rates to prevent overheating and control inflation. Rising interest rates lead to higher yields on new bonds, making existing bonds with lower yields less attractive. As a result, bond prices may decline, especially for long-term bonds. Additionally, when the economy is strong and businesses are performing well, investors may prefer to allocate more of their funds to the potentially higher returns offered by the stock market rather than fixed income investments. The same reasoning applies to the fact that there will be lower demand for safe haven assets such as government bonds as investors are more confident about the overall economic outlook. Consequently, demand for such bonds may decrease, putting downward pressure on prices.

Commodities

A strong economy typically translates to higher industrial production and increased consumer spending. As a result, the demand for commodities, especially industrial metals like copper and steel, tends to rise due to their use in manufacturing and construction. Economic growth also leads to higher energy consumption. Therefore, oil and natural gas demand often increases, pushing their prices upward. Rising inflation expectations is a result of a growing economy, and this may lead investors to turn to precious metals (such as gold and silver) as a hedge against currency devaluation and market uncertainties. Finally, agricultural commodities (such as wheat and corn) may also witness a rise due to increased consumer spending on food products.

Currencies

A robust economy typically attracts foreign investment and boosts demand for the domestic currency. International investors seek higher returns in a strong economy, leading to increased demand for assets denominated in that currency. As demand for the domestic currency rises, its value appreciates relative to other currencies in the foreign exchange market. Central banks may raise interest rates to manage the rising inflation and ensure economic stability. Higher interest rates attract foreign investors seeking better returns, driving up demand for the domestic currency and pushing its value higher. Similarly, a strong economy often leads to increased exports, as domestic goods and services become more competitive globally. This can improve the country's trade balance and contribute to a stronger currency.



These relationships are more complex in real life, as other factors such as political stability and weather events enter into the equation. It is not uncommon to see violations and structural breaks of these correlations and market logic. They may be used as rules of thumb but never as foolproof conditions.

The concept of *economic intuition* refers to the rationality of selecting features to predict a dependent variable. A basic example of applying economic intuition to predict stock market returns is to select companies' earnings and treasury yields as independent variables. Economic intuition is recommended when creating sophisticated trading algorithms based on exogenous and fundamental factors.

Some experts argue that technical indicators may not be enough to deliver a full picture of what to expect next, but others point to the fact that fundamental measures cannot be used in the short term, and sometimes long-term data is not accessible or sufficient.

News Interpretation

News interpretation is the process of analyzing and understanding how news events, economic indicators, geopolitical developments, and other relevant information impacts financial markets. Before exploring how to analyze news, it is important to refresh your knowledge of economic indicators:

Gross domestic product (GDP)

This measure represents the total monetary value of all goods and services produced within a country's borders during a specific period, typically measured on a quarterly or annual basis. GDP is used as a measure of the overall economic health and performance of a nation.

Unemployment rate

This measure calculates the percentage of the labor force that is unemployed and actively seeking employment. It is a key metric used to assess the health of the job market and overall economic conditions. The unemployment rate is typically reported on a monthly basis by government agencies. A lower unemployment rate is generally considered positive as it indicates a healthier job market, higher economic activity, and more opportunities for job seekers. Conversely, a higher unemployment rate can be a sign of economic challenges and underutilization of the labor force. Policymakers closely monitor the unemployment rate to gauge the effectiveness of their economic policies and make informed decisions to address employment-related issues.

Inflation rate

Simply put, inflation is the overall increase in the general price level of goods and services in an economy over a period of time. When inflation occurs, each unit of currency buys fewer goods and services than it did previously. In other words, inflation erodes the purchasing power of money. Inflation is commonly measured using various price indices, such as the consumer price index (CPI). It is typically calculated on a year-on-year basis. You saw the CPI in [Chapter 3](#).

Central bank policy rates

Also known as *benchmark rates*, they represent the interest rates set by a country's central bank to influence and control the overall economic activity and inflation within the country. These rates serve as a primary tool of the monetary policy. The most common central bank policy rate is the *overnight* or *policy* rate, which is the interest rate at which commercial banks can borrow or lend money to the central bank on an overnight basis. This rate is crucial because it influences various other interest rates in the economy, including those charged on loans and earned on savings accounts. Central banks use changes in policy rates to achieve specific economic objectives, such as controlling inflation, stimulating growth, and maintaining price stability.

Consumer confidence indices

These indices measure the level of optimism or pessimism among consumers regarding their current and future economic prospects. These indices are designed to gauge consumer sentiment, which can provide insights into consumer spending behavior and overall economic health. Consumer confidence indices are usually based on surveys conducted among a representative sample of households.



Another important economic indicator is the ISM PMI, an indicator discussed in [Chapter 9](#).

As a trader, news interpretation involves assessing the significance of news items and determining their potential effects on asset prices. Consider the following points when consuming news:

- Different news items can affect various asset classes differently. For instance, positive economic data might boost stocks but lead to a decline in safe-haven assets like gold. Understanding the correlations and inverse relationships between news events and assets is crucial.
- News can shape market sentiment, leading to either bullish or bearish trends. Traders must gauge how news is perceived by market participants and how it may influence their behavior.
- Some news has an immediate impact on markets, causing sudden price movements, while other news may have a more gradual and prolonged effect. Traders should be aware of the time sensitivity of news to optimize their trading strategies.

- In some cases, the market has already priced in expectations of certain news events. Traders need to compare the actual news release with market expectations to determine whether the news is a surprise or not.



Staying informed through reputable financial news outlets (such as the *Financial Times* and Bloomberg) and conducting thorough research can help traders make better-informed decisions in a dynamic and ever-changing market environment.

Risk Management

Finding a predictive and profitable trading strategy is what will make you money, but a sound risk management system is what will allow you to keep it. In traditional finance, the most basic form of risk management is *diversification*, which means to allocate your funds into different uncorrelated asset classes so that you spread out the risk of capital depreciation.

Another example of basic risk management is simply the avoidance of scams that may plague the retail trading community every now and then.

Nowadays, there is a huge number of asset managers, signal providers, and copytrading algorithms that promise high and consistent returns. The abundance of these online services begs the question as to whether the quality of the investment is worth it or not. There are three simple rules to follow when you assess a trading service:

- *Guaranteed returns* and *scam* are two interchangeable concepts. Avoid falling into this trap.
- Even the slightest vagueness in a track record is a major red flag. Ask for transparent and interpretable results backed by third-party auditors.
- The overwhelming majority of track records are fake, and the results are mostly based on faulty backtests or selection bias. You have to always be skeptical.

Another example of risk management is position sizing. *Position sizing* refers to the process of determining the appropriate amount of capital to allocate to a particular trade or investment. It involves calculating the number of shares, contracts, or lots to trade based on various factors such as risk tolerance, account size, and market conditions. Proper position sizing is crucial for risk management and helps traders avoid excessive losses. Smart trading algorithms have this functionality embedded in them when they trade so that the positions are optimized based on a number of factors such as volatility and expected return.

There are several techniques and methods for position sizing, such as the following:

Fixed monetary amount

This method involves allocating a fixed monetary amount of your trading capital to each trade. For example, you may decide to risk \$250 of your account on each trade. This approach therefore assumes equal weights for every trade.

Fixed fractional

With this technique, you allocate a fixed fraction of your trading capital to each trade. For instance, you may decide to risk 1% of your account on each trade. As your account size changes, the position size adjusts accordingly.

Kelly criterion

The Kelly criterion is a mathematical formula that calculates the optimal position size based on the probability of winning and losing trades. It takes into account the trader's edge (probability of success) and the payoff ratio. The formula helps determine the percentage of capital to allocate to maximize long-term growth while minimizing the risk of ruin.

Hit ratio method

The hit ratio method is a technique I use that takes a momentum approach with regard to winning and losing streaks. It assumes that winning must be rewarded with bigger positions while losing must be penalized with smaller positions. The better the hit ratio, the bigger the positions get and vice versa.

Basics of Risk Management

Risk management does not have to be complex. You can create a solid risk management system by using the few simple techniques given in this section.

Stops and targets

A *stop-loss order* (stop) is an automatic order set during trade initiation to ensure a minimal predetermined loss. For example, if you buy a few contracts on gold at \$1,500 in anticipation that it will go to \$2,000, you may place your stop-loss order at \$1,250 so that if the price drops to \$1,250, you limit your losses to no more than \$250. At the same time, your anticipation of selling when the market price reaches \$2,000 is known as a *take-profit order* (target).



The most basic risk management system for any trade is setting proper stops and targets so that you are framing your expectations and limiting your risks.

There is one rule that you must never violate: always place stops and targets. Algorithms easily allow you to automate this process.

Trailing stops

A *trailing stop* is a dynamic stop-loss order that follows the market price whenever it goes in the expected direction, thus ensuring less loss and at some point in time ensuring a no-loss trade. The best way to understand the trailing stop order is through an example.

Suppose you go long on EURUSD at \$1.0000 to target \$1.0500. You set your stop-loss order at \$0.9900 to limit your losses. Now suppose two days have passed and the current market price is at \$1.0230, and you want to ensure that your trade never loses money. The solution to this is a trailing stop order where you move your stop from \$0.9900 to \$1.0000 so that, in the worst-case scenario, when the market drops back to \$1.0000 you close out and break even.

Another way of ensuring at least some profit is to move the stop to \$1.0100, thus ensuring a profitable trade no matter what. Of course, the closer you move your stop to the current market price, the more likely you are to be stopped out at a profit.



It is always a good idea to use trailing stops with trend-following strategies as they allow you to ride the trend.

Economic calendar

The *economic calendar* is a form of risk avoidance as it shows the important news releases that are expected to have an impact on the markets. **Table 12-1** shows a hypothetical example of an economic calendar on a certain day.

Table 12-1. Economic calendar

Time	Country	Event	Impact	Previous value	Expected value	Actual value
9:00 AM	United Kingdom	CPI	High	1.00%	1.20%	1.10%
11:30 AM	Germany	Core CPI	High	0.50%	0.75%	0.75%
4:30 PM	USA	Initial jobless claims	Low	232,000	215,000	229,000
7:30 PM	USA	Interest rate decision	High	1.50%	1.50%	TBA

Some traders trade based on the news, and therefore they like to trade ahead of news releases or seconds after them to profit from volatility. Risk management-wise, this is not recommended, as fluctuations are random around news events, which can surprise the markets from time to time.

The best thing to do is to avoid trading around the times of an important release that has historically caused some extreme variations. Examples may include political announcements, GDP numbers, and FOMC meetings.



FOMC stands for *federal open market committee*. This committee oversees the open market treasury operations in the United States and makes decisions about the interest rate.

Behavioral Finance: The Power of Biases

Behavioral finance, a field stemming from behavioral economics, attempts to explain market anomalies and traders' actions. By having a deep understanding of behavioral finance, you will better understand why the market reacts the way it does, especially around certain events and levels.

Financial markets are composed of the actions and reactions of different human and robotic participants, which makes for a psychological and quantitative stew. This explains the low signal-to-noise ratio—in other words, why it is difficult to accurately forecast the market on a regular basis. These actions and reactions are also referred to as *biases*, and they are human shortcomings in response to certain events. Biases are the main protagonists of this section, and they are divided into two categories:

Cognitive biases

These biases come from a lack of knowledge. Cognitive biases generally involve incorrect conclusions based on wrong market assumptions or bad research.

Emotional biases

These biases are mostly feelings driven and are impulsive in nature. They are not caused by a lack of education but by a lack of self-control and self-management.

Cognitive biases

This subsection lists some of the most common cognitive biases and their market impact and offers recommendations on how to avoid them:

Conservatism bias

This occurs when a market participant is slow to react to new information and places too much weight on base rates. It is a type of failure to adapt. The way to deal with this bias is to force yourself to be skeptical of the basic analysis and to always be dynamic and ready for change. The market does not always behave as it did in the past since it is forward looking.

Confirmation bias

This occurs when the trader focuses on the type of information that benefits their ongoing position and dismisses the type of information that is not favorable to their position. This is by far one of the most common biases and is actually a normal state of mind that leads to overconfidence over time. The best way to remedy this is to remain impartial and neutral, which is easier said than done.

Another way is to automate the decision-making process through scorecards highlighting the attractiveness (or not) of the analyzed underlying assets. Humans in general suffer from this bias—it is not just finance specific.

Illusion of control bias

This occurs when the trader overestimates their ability to control the trade outcome. Mainly caused by a streak of winning trades, this bias can lead to concentrated positions due to a sense of power over the invested asset. Markets are composed of a huge number of participants with trillions of invested dollars and as such are unlikely to be impacted by any single person (there are a few, very rare exceptions involving small and illiquid assets). The way to remove the illusion of control is to stay focused and humble and understand that you are facing a semirandom environment that changes its dynamics and drivers every day.

Hindsight bias

This occurs when a trader overestimates their past accuracy and can lead to excessive risk taking. It is easy to look at past charts and conclude that the subsequent direction was obvious. Most backtests include a form of hindsight bias since conditions were created at the end of the analyzed period. Market technicians overestimate their abilities when they see that some techniques worked well in the past but fail to take into account the environment of the tested period. Also, some configurations do not look the same when they're happening as when they are complete. Hindsight bias can be hard to cure, but the best way is to take into consideration the variables that were present during the analyzed period to simulate the past environment in a more realistic way.

Anchoring bias

This occurs when the trader's opinion is anchored to a certain base point and fails to change to incorporate new information. As I have mentioned, the analyst or trader must maintain a dynamic and open mindset. The best way to cure this bias is to stay up to date with regard to new information and pieces of data.

Availability bias

This occurs when the trader selects positions based on how easily they can retrieve memories of them. This means that familiar assets are more attractive than nonfamiliar assets, which is a wrong assumption as opportunities may come from any type of market. It is a type of mental shortcut where the trader does not expend much effort in research. To cure this bias, you must practice full due diligence before selecting a universe of investable assets. Do not just trade the EUR-USD because you are familiar with it: expand your horizons.

Loss aversion bias¹

This occurs when the pain of losing is greater than the joy of winning. It is by far the most common bias. Humans are known to prefer not to lose money than to gain money, as was demonstrated in “Prospect Theory: An Analysis of Decision Making Under Risk” by Daniel Kahneman and Amos Tversky (1977). Loss aversion could lead to decreased risk taking and therefore decreased expected return. However, the most significant effect is on the stop levels. Loss-averse humans do not want to accept that they are losing money and will see a losing position as an ongoing position and, therefore, prefer to wait until it turns positive. This is highly dangerous, as leaving positions without stop-loss levels could lead to disastrous results. In addition, some people close out of winning positions too quickly in fear that they will turn negative (a form of fear of regret). The best way to handle loss aversion is to automate the risk management process and to respect the stop-loss and target orders established at the initiation of the trade.

Emotional biases

This subsection lists some of the most common emotional biases and their impact. As a reminder, while cognitive biases are related to a lack of knowledge, emotional biases are related to psychological traits:

Overconfidence bias

This occurs when a trader enjoys a winning streak and believes it is due to their superior ability to trade the markets, which leads to holding concentrated positions and excessive trading. A good streak will come to an end, and thus the trader must always follow procedures and ensure that they do not stray from the strategy.

Regret-aversion bias

This refers to staying in low-risk investments out of fear. This really is all about the risk profile of the trader. There is no right or wrong answer, but the fear of regret can make the trader lose out on interesting opportunities. You should take risks to make money, but only calculated risks.

Endowment bias

This occurs when the trader believes that the owned assets are more valuable than the ones that are not owned. This may hamper the trader's opportunities and cause them to be limited to the assets they already own, even if they decay over time. The market has opportunities everywhere, and participants must always be on the lookout for the next big thing.

¹ This is also considered an emotional bias.

Summary

This chapter discussed miscellaneous trading topics that are helpful to guide you in your trading journey. Trading must also encompass subjective opinions so that objective ones are created. Risk management is a crucial aspect of trading and investing that involves identifying, assessing, and mitigating potential risks to protect capital and minimize losses. Its primary goal is to preserve capital while aiming for consistent, long-term profitability.

By finishing this chapter, you have gone through the book and have understood how to develop machine and deep learning algorithms that aim to forecast price returns and other time series. The journey does not stop here; you must continue to experiment with the different techniques until you find a strategy that suits your risk-reward profile the most.

You should also acknowledge how far you have come by completing this book. Writing is a solitary endeavor, but it's your engagement with the words on these pages that brings the words to life. Your dedication to the material and your willingness to embark on this adventure with me mean the world to me.

I hope this book has inspired or enlightened you in some way. It was written with love, passion, and a deep desire to simplify complex concepts. Your presence in this narrative has made it all the more meaningful.

Index

A

activation functions, 217-223
AdaBoost regression, 207-209
adaptive moment estimation (Adam), 226
addition rule, 22
ADF (augmented Dickey–Fuller) test, 72-76
adjacent line, 102
adversarial training, 276
algebra, defined, 81
algorithms
 data science and, 10-13
 defined, 10
alpha trading, 136
alternative investment classes, 16
anchoring bias, 334
ANNs (see artificial neural networks)
anonymous (lambda) function, 272
arbitrage, 11
arithmetic operators, in Python, 156
arrays, in Python, 155
artificial neural networks (ANNs), 215-231
 activation functions, 217-223
 backpropagation, 224-225
 layers, 216
 multilayer perceptrons, 227-231
 optimization algorithms, 226
 regularization techniques, 226
audio data, 4
augmented Dickey–Fuller (ADF) test, 72-76
autocorrelation, 66
autoregressive models, 192
availability bias, 334

B

backpropagation, 224-225, 228
bagging, 205
batch normalization, 271-275
batch size hyperparameter, 225
bearish markets, 14
behavioral finance, 333-335
 cognitive biases, 333-335
 emotional biases, 335
Bellman equation, 280
benchmark rates, 329
bias
 cognitive, 333-335
 in machine learning models, 211-214
biased model, 186
biases, behavioral finance and, 333-335
Bitcoin, predicting volatility using deep learning, 308-317
blockchain, 308
bonds, 15
Booleans, in Python, 155
boosting
 AdaBoost regression, 207-209
 XGBoost regression, 209-211
bullish markets, 14
business intelligence, defined, 12

C

calculus, 105-132
 defined, 81
 derivatives, 114-123
 integrals and the fundamental theorem of calculus, 114-123
 limits and continuity, 105-114

linear algebra versus, 81
optimization, 128-132

candlestick charts
charting analysis, 137-142
moving averages, 143-144

candlestick patterns, 147

Cartesian coordinate system, 82

categorical data, 4

central bank policy rates, 329

central limit theorem, 26

central tendency
defined, 36
measures of, 36-79
measures of shape, 45-54
measures of variability, 41-45

chain rule, 122

charting analysis, 137-142

classic price patterns, 147

cluster sampling, 25

CNNs (convolutional neural networks), 243-246

coefficient of determination (R^2), 242

cognitive biases, 333-335

comments, in Python, 153

commercial traders, 291

Commitments of Traders (COT) report
algorithm 1: indirect one-step COT model, 297-299
algorithm 2: MPF COT direct model, 299-300
algorithm 3: MPF COT recursive model, 301-303
using COT data to predict long-term trends, 291-304

commodities
defined, 16
market drivers, 327

comparison operators, in Python, 156

concatenation, 167

conditional probability, 22

conditional statements, 157

confidence intervals, 26

confirmation bias, 333

conjugate, 113

conservatism bias, 333

constant predictions, 321

constants, 153

consumer confidence indices, 329

consumer price index (CPI), 35-79

continuation price patterns, 147

continuity, limits and, 105-114

continuous distributions, 46

continuous functions, 112

continuous random variable, 20

continuous retraining, 256-258

contrarian indicators, 143

control flow, in Python, 157-159

convolutional layers, 243

convolutional neural networks (CNNs), 243-246

correlation, 3, 63-70

COT report (see Commitments of Traders report)

covariance, 64

CPI (consumer price index), 35-79

credit scoring, data science's role in, 17

critical points, 130

cross validation, 259-261
sampling and, 25
time series cross validation, 259-261

cryptocurrency, 308-317

currency markets
defined, 16
market drivers, 327

D

data

alternative methods for importing, 188
defined, 1
types and structures, 1-10
visualizing, 54-62

data analysis, 35-80

data collection, in Python, 155

data engineers, data scientists versus, 13

data interpretation, 12

data prediction, 12

data science
algorithms, 10-13
applications in finance, 17-18
defined, 4
Python for, 151-175
steps in, 4-9
visualizing data, 54-62

data scientists, data engineers versus, 13

data storage, 10

data structures, numpy and pandas, 166-169

data types, Python, 154-156

dataframe, in Python, 155

- DDQN (Double Deep Q-Network), 285
dealers (commercial traders), 291
death cross, 144
decision tree regression, 203-205
decision trees
 defined, 20
 random forest regression, 205-207
 XGBoost regression, 209-211
deep learning (generally)
 essential probabilistic methods for, 19-34
 machine learning versus, 215
 time series prediction, 18
deep reinforcement learning, 284-290
 basics, 284-290
definite integrals, 126
deflation, 52
derivatives (calculus), 114-123
derivatives (financial instrument), 14
descriptive statistics
 correlation, 63-70
 defined, 35
 measures of shape, 45-54
 measures of variability, 41-45
 regression analysis and statistical inference, 70-77
 stationarity, 70-77
 visualizing data, 54-62
dictionaries, in Python, 155
differencing, 181
differential calculus, 105
digital currency, 308-317
discontinuous functions, 112
discrete distributions, 46
discrete random variable, 20
disinflation, 52
diversification, 330
dividend, 2
dividend yield, 2
dot product method, 90
Double Deep Q-Network (DDQN), 285
double top pattern, 147-148
DropConnect regularization technique, 275
dropout (regularization technique), 227
dummy regression, 180-183
- E**
early stopping (regularization technique), 227
economic calendar, 332
economic intuition
 defined, 328
 market drivers and, 326-328
elimination (algebra), 100
emotional biases, 335
endowment bias, 335
ensemble learning, 205
entropy
 calculating, 32
 defined, 30
epoch, 224
epsilon
 gamma versus, 283
 in reinforcement learning, 282
Euler's number, 103
exception handling, in Python, 163-165
expanding window cross validation, 260
expected value of a random variable, 24
experience replay, 284
exponent of a number, 31
exponential derivatives, 121
- F**
factoring, 112
fear index, 5
fear of missing out (FOMO), 312
Federal Open Market Committee (FOMC), 330
feedforward neural network, 227
Fibonacci retracements, 142
filters, 324
financial fraud detection, 17
financial markets, 14-17
financial time series, 18
fitting problem, 211-214
fixed income market
 defined, 15
 market drivers, 327
FOMC (Federal Open Market Committee), 330
FOMO (fear of missing out), 312
forecasting
 defined, 12
 of market direction, 17
forecasting threshold, 254-256
forward propagation, 224
fractional differentiation, 249-253
fraud detection, 17
functions, Python, 159-162
fundamental analysis, 136
fundamental theorem of calculus, 127-128
funds (noncommercial traders), 291

FX (foreign exchange) market

defined, 16

market drivers, 327

G

gamma, epsilon versus, 283

Gaussian (normal) distribution, 46

GD (gradient descent), 197, 226

GDP (gross domestic product), 328

global maximum, 130

global minimum, 129

going long (long position), 14

going short (short position), 14

golden cross, 144

gradient boosting, 209-211

gradient descent (GD), 197, 226

gross domestic product (GDP), 328

H

hedgers (commercial traders), 291

hedging, 15

hind sight bias, 334

Hinton, Geoffrey, 216

hyperbolic functions, defined, 102

hyperinflation, 52

hypotenuse, 102

hypothesis testing, 20, 27-30

I

identity matrix, 91

illusion of control bias, 334

in-sample set, 25

indefinite integrals, definite integrals versus, 126

independent events, 23

indicator analysis, 143-146

moving averages, 143-144

relative strength index, 145-146

infinity, 111

inflation, 51

inflation rate, 328

inflection points, 130

information gain

calculating, 33

defined, 30

information theory, 20, 30-34

integral calculus, 105

integrals, 124-127

integration, 124

interquartile range (IQR), 53

invisible hand, 142

irrational numbers, 103

J

joint probability, 22

K

k-fold cross validation, 259

k-nearest neighbors (KNN) regression algorithm, 200-202

kurtosis, 50

Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test, 74-76

L

L1/L2 regularization (weight decay), 275

lambda function, 272

Lasso regression, 192

leaky ReLU (rectified linear unit) activation function, 222

learning rate hyperparameter, 225

leptokurtic distribution, 51

leveraged money (noncommercial traders), 291

libraries, Python, 159-162

limits, 105-114

linear algebra

calculus versus, 81

linear equations, 92-96

systems of equations, 96-101

trigonometry, 101-105

vectors and matrices, 82-92

linear correlation, 64

linear equations, 92-96

linear regression algorithm, for time series prediction, 190-194

linear regression, defined, 77

linear scale, logarithmic scale versus, 310

liquidity, defined, 16

lists, in Python, 155

local maximum, 129

local minimum, 129

logarithm, defined, 31

logarithmic derivatives, 121

logarithmic scale, linear scale versus, 310

logical operators, in Python, 157

long position, 14

long short-term memory (LSTM), 235-242
for indirect one-step COT model, 297-299
for MPF COT direct model, 299-300
loops, in Python, 158-159
loss aversion bias, 335
loss function, 130, 184
LSTM (see long short-term memory)

M

machine learning (for time series prediction), 177-214
AdaBoost regression, 207-209
decision tree regression, 203-205
framework, 177-187
k-nearest neighbors (KNN) regression algorithm for, 200-202
linear regression algorithm, 190-194
machine learning models for, 190-211
overfitting and underfitting, 211-214
random forest regression, 205-207
stochastic gradient descent regression for, 197-200
support vector regression algorithm for, 194-197
XGBoost regression, 209-211
machine learning, deep learning versus, 215
MAE (mean absolute error), 184
market direction, forecasting, 17
market drivers, 325-330
economic intuition and, 326-328
news interpretation, 328-330
market efficiency, 136
market microstructure, 16
matrices, 89-92
matrix transposing, 91
maximal information coefficient (MIC), 68-70
maximum drawdown, 187
McCulloch, Warren, 215
mean
as central tendency measure, 36
defined, 7
mean absolute error (MAE), 184
mean squared error (MSE), 185
measures of central tendency, 36-40
measures of shape, 45-54
measures of variability, 41-45
median, defined, 38
mesokurtic distribution, 51

MLPs (multilayer perceptrons), 227-231, 271-276
mode, defined, 40
model evaluation, 184
modules, in Python, 159
moving averages, 143-144
MPF (multiperiod forecasting), 261-270
MSE (mean squared error), 185
multicollinearity, 304
multilayer perceptrons (MLPs), 227-231, 271-276
multiperiod forecasting (MPF), 261-270
multiple linear regression model, 192
mutual exclusivity, 21
mutual information, 33

N

naive forecasting, 193
natural exponent function, 103-104
natural language processing (NLP), 18
nearest neighbors regression algorithm, 200-202
negative correlation, 3
negative skew, 48
net return, 186
neural networks, 18
news interpretation, 328-330
NLP (natural language processing), 18
noncommercial traders, 291
nonreportable traders, 292
normal distribution, 46
numerical data, 3
numerical data types (Python), 154
numpy, data structures in, 166-169

O

OLS (ordinary least squares) method, 190
one-tailed test, 28
operators, in Python, 156
opposite line, 102
optimization (calculus), 128-132
optimizers (optimization algorithms), 226
options, 15
ordinary least squares (OLS) method, 190
out-of-sample set, 25
overconfidence bias, 335
overestimation bias, 286
overfitting, 211-214

P

p-value, 30
pandas, data structures in, 166-169
pattern recognition, 147-149
Pearson correlation coefficient, 64
perceptrons, 215, 227
Pitts, Walter, 215
pivot points, 142
platykurtic distribution, 51
policy, reinforcement learning, 282
pooling layers, 243
position sizing, 330
positive correlation, 3
positive skew, 47
power rule, 117-118
predictive analytics, 12
price discovery, 16
probabilistic methods, 19-34
 information theory basics, 30-34
 probabilistic concepts, 20-24
 probability basics, 19-30
 sampling and hypothesis testing, 25-30
probability, 19-30
probability distribution functions
 defined, 20
 measure of shape, 45
 random variables and, 21
product rule, 118
profit factor, 186
Python, 151-175
 basic operations and syntax, 153-157
 basics for data science, 151-175
 control flow, 157-159
 data structures in numpy and pandas,
 166-169
 data types, 154-156
 downloading, 151-153
 exception handling and errors, 163-165
 importing financial time series in, 170-175
 libraries and functions, 159-162
Python interpreter, 152

Q

Q-learning, 280
Q-table (quality table), 279
quantiles, 52-53
quantitative analysis, 136
quartiles, 52
quotient rule, 120

R

random forest regression, 205-207
random variable, defined, 20
reciprocal, 93
rectified linear unit (ReLU) activation function,
 220-222
recurrent neural networks (RNNs), 232-235
recursive model
 MPF COT recursive model, 301-303
 multiperiod forecasting, 261-267
regression analysis, 70-77
regret-aversion bias, 335
regularization
 applying to MLPs, 271-276
 techniques, 226
reinforcement learning, 12
 algorithms, defined, 12
 deep reinforcement learning basics, 284-290
 defined, 277
 intuition of, 278-283
 main elements of, 278
relative strength index (RSI), 145-146
reliability factor, 26
ReLU (rectified linear unit) activation function,
 220-222
replay memory, 284
resistance level, 138
retraining, continuous, 256-258
returns, defined, 181
reversal price patterns, 147
reward function, 279
ridge recession, 192
risk management, 330-335
 basics, 331-333
 behavioral finance, 333-335
 data science's role in, 17
 economic calendar, 332
 stops and targets, 331
 trailing stops, 332
risk-free rate, 188
RMSE (root mean squared error), 185
RMSprop (root mean square propagation), 226
RNNs (recurrent neural networks), 232-235
rolling window cross validation, 259
root mean square propagation (RMSprop), 226
root mean squared error (RMSE), 185
Rosenblatt, Frank, 215
RSI (relative strength index), 145-146

S

sampling
 basics, 25-27
 defined, 25
sampling error, 26
scalar, 88
scatterplots, 54
serial correlation (autocorrelation), 66
set, in Python, 155
SGD (stochastic gradient descent) algorithm,
 197-200, 226
shape, measures of, 45-54
Sharpe ratio, 187
short position, 14
sigmoid activation function, 217
significance level, 26
simple linear regression model, 192
simple random sampling, 25
slope, 86, 115
smoothed moving average, 146
Spearman's rank correlation, 65-66
Spyder interface, 152
stagflation, 52
standard deviation, 43
standard error, 27
stationarity, 70-77
statistic, defined, 26
statistical analysis, stationarity and, 70-77
statistical inference, 70-77
stochastic gradient descent (SGD) algorithm,
 197-200, 226
stock market
 defined, 15
 market drivers, 326
stop-loss order, 331
stratified sampling, 25
strings, in Python, 155
substitution (algebra), 99, 112
supervised learning algorithms, 12
support level, 138
support vector regression (SVR) algorithm,
 194-197
syntax, defined, 153
systematic sampling, defined, 25
systems of equations, 96-101

T

t-distribution, 27
take-profit order (target), 331

target (take-profit order), 331
technical analysis, 135-149
 charting analysis, 137-142
 indicator analysis, 143-146
 pattern recognition, 147-149
technical indicators, as inputs, 304-308
temporal convolutional neural networks,
 243-246
testing sample, 25
text data, 4
time frame, 172
time series cross validation, 259-261
time series prediction (deep learning algo-
 rithms for), 215-247
 challenges when applying algorithms, 247
 long short-term memory, 235-242
 multiperiod forecasting, 261-270
 neural networks, 215-231
 recurrent neural networks, 232-235
 temporal convolutional neural networks,
 243-246
time series prediction (deep learning tech-
 niques and methods for), 249-276
 applying regularization to MLPs, 271-276
 continuous retraining, 256-258
 forecasting threshold, 254-256
 fractional differentiation, 249-253
 time series cross validation, 259-261
time series prediction (deep reinforcement
 learning for), 277-290
time series prediction (machine learning for),
 177-214
 AdaBoost regression, 207-209
 decision tree regression, 203-205
 framework, 177-187
 k-nearest neighbors (KNN) regression algo-
 rithm for, 200-202
 machine learning models for, 190-211
 overfitting and underfitting, 211-214
 overfitting and underfitting of machine
 learning model for, 211-214
 random forest regression, 205-207
 stochastic gradient descent regression for,
 197-200
 support vector regression algorithm for,
 194-197
 XGBoost regression, 209-211
time series, importing in Python, 170-175
timing patterns, 147

trading, 14-17
trading evaluation, 184
training sample, 25
training, real-time visualization of, 317-323
transformations (technical indicators), 304-308
transposing (matrices), 91
trend-following indicators, 143
trigonometric functions, defined, 102
trigonometry, 101-105
Tukey's hinges, 53
tuple, in Python, 155
two-tailed test, 28
type I/type II error, 29

U

unconditional probability, 22
undefined function, 106
underfitting, 212
unemployment rate, 328
unit root, 72
unsupervised learning algorithms, 12

V

vanishing gradient problem, 219, 232
variability, measures of, 41-45

variables (Python), 153
variance
 defined, 41
 in machine learning models, 212
vectors, 82-88
visual data, defined, 4
visualization
 of data, 54-62
 of training, 317-323
VIX (volatility index), 5-9
volatility (standard deviation), 43

W

weight constraints regularization technique,
 276
weight decay (L1/L2 regularization), 275

X

x-axis, 82
XGBoost regression, 209-211

Y

y-axis, 82

About the Author

Sofien Kaabar is a financial author, trading consultant, and institutional market strategist specializing in the currencies market with a focus on technical and quantitative topics. Sofien's goal is to make technical analysis objective by incorporating clear conditions that can be analyzed and created with the use of technical indicators that rival existing ones.

Having developed many successful trading algorithms, Sofien is now sharing the knowledge he has acquired over the years to make it accessible to everyone.

Colophon

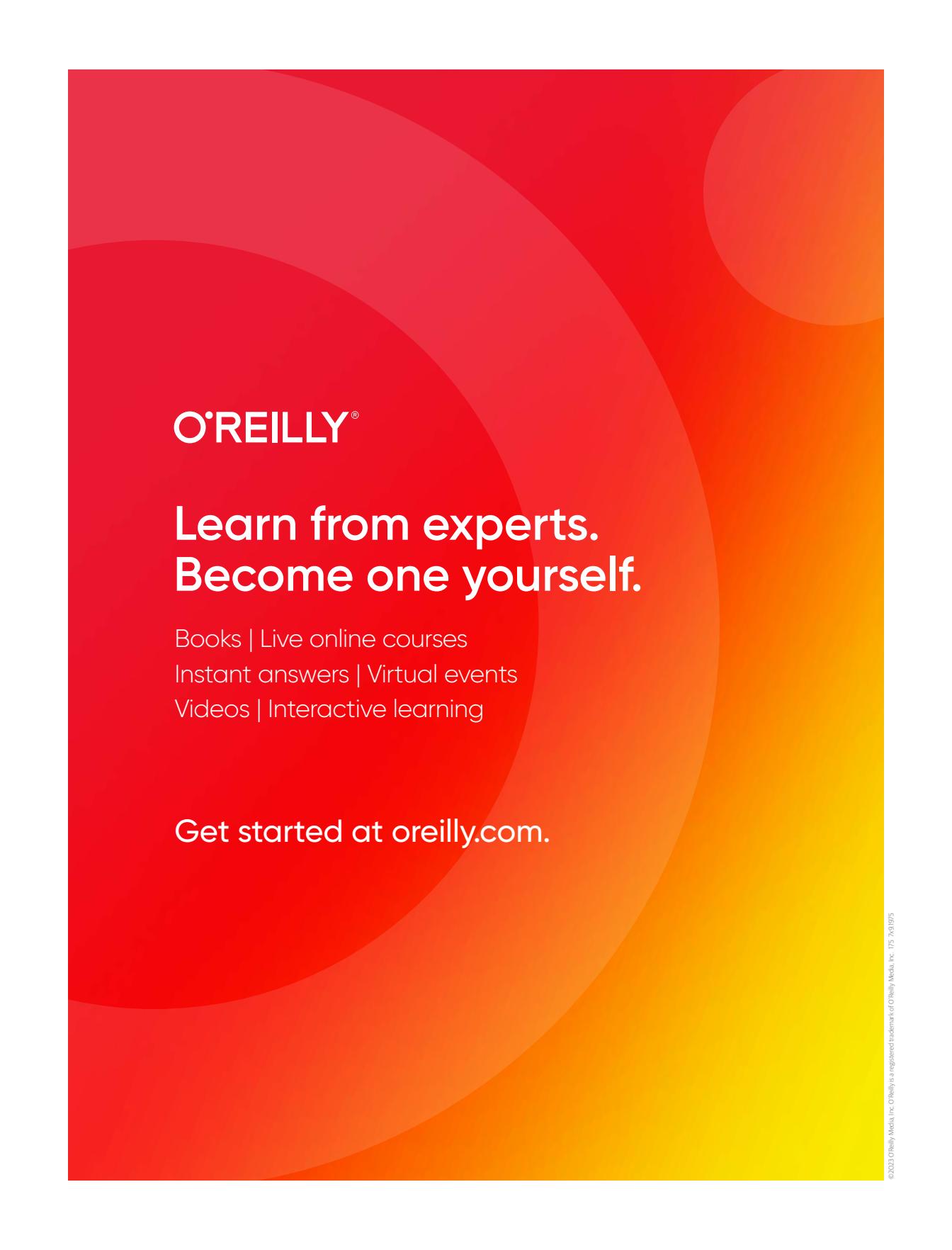
The animal on the cover of *Deep Learning for Finance* is a fantail goldfish (*Carassius auratus*). Fantail goldfish are a common type of fancy goldfish. They cannot be found in the wild; people bred goldfish in tanks to achieve specific traits over several generations.

The appearance of fantail goldfish is defined by their double tail, which includes two top and bottom caudal fins. They have short, egg-shaped, bulbous bodies and develop telescope eyes around the age of 6 months. Their bodies are typically an orange or white color, but they can also be shades of red or multicolored. They grow between 6 and 8 inches long, and they can live anywhere from 5 to 15 years depending on care.

Fancy goldfish are omnivorous and should be fed a highly nutritious diet to avoid buoyancy disorders. If using pellets, it is important to select a size that is easily digestible. Because their stomachs are not acidic, these fish rely heavily on grinding their molars to mash up food for digestion. Other feeding options include live or frozen food, such as brine shrimp, daphnia, and bloodworms.

While fantail goldfish are not endangered, many of the animals on O'Reilly covers are; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Dover's Animals*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

Learn from experts. Become one yourself.

Books | Live online courses
Instant answers | Virtual events
Videos | Interactive learning

Get started at oreilly.com.