

THE PYTHON SERIES

INTRODUCTION TO QUANTITATIVE SOCIAL SCIENCE WITH PYTHON



WEIQI ZHANG
DMITRY ZINOVIEV

A **Chapman & Hall** Book



CRC Press
Taylor & Francis Group

Introduction to Quantitative Social Science with Python

Departing from traditional methodologies of teaching data analysis, this book presents a dual-track learning experience, with both Executive and Technical Tracks, designed to accommodate readers with various learning goals or skill levels. Through integrated content, readers can explore fundamental concepts in data analysis while gaining hands-on experience with Python programming, ensuring a holistic understanding of theory and practical application in Python.

Emphasizing the practical relevance of data analysis in today's world, the book equips readers with essential skills for success in the field. By advocating for the use of Python, an open-source and versatile programming language, we break down financial barriers and empower a diverse range of learners to access the tools they need to excel.

Whether you're a novice seeking to grasp the foundational concepts of data analysis or a seasoned professional looking to enhance your programming skills, this book offers a comprehensive and accessible guide to mastering the art and science of data analysis in social science research.

Key Features:

- Dual-track learning: Offers both Executive and Technical Tracks, catering to readers with varying levels of conceptual and technical proficiency in data analysis.
- Includes comprehensive quantitative methodologies for quantitative social science studies.
- Seamless integration: Interconnects key concepts between tracks, ensuring a smooth transition from theory to practical implementation for a comprehensive learning experience.
- Emphasis on Python: Focuses on Python programming language, leveraging its accessibility, versatility, and extensive online support to equip readers with valuable data analysis skills applicable across diverse domains.

WeiQi Zhang is an Associate Professor at Suffolk University. He teaches courses on political science and data analytics, and he is passionate about bridging social sciences and data science.

Dmitry Zinoviev is a Professor of Computer Science at Suffolk University. His academic interests include computer modeling and simulation, complex networks, and the integration of computational methods into traditionally non-quantitative fields such as the humanities and social sciences.

Chapman & Hall/CRC

The Python Series

About the Series

Python has been ranked as the most popular programming language, and it is widely used in education and industry. This book series will offer a wide range of books on Python for students and professionals. Titles in the series will help users learn the language at an introductory and advanced level, and explore its many applications in data science, AI, and machine learning. Series titles can also be supplemented with Jupyter notebooks.

Image Processing and Acquisition using Python, Second Edition

Ravishankar Chityala and Sridevi Pudipeddi

Python Packages

Tomas Beuzen and Tiffany-Anne Timbers

Statistics and Data Visualisation with Python

Jesús Rogel-Salazar

Introduction to Python for Humanists

William J.B. Mattingly

Python for Scientific Computation and Artificial Intelligence

Stephen Lynch

Learning Professional Python Volume 1: The Basics

Usharani Bhimavarapu and Jude D. Hemanth

Learning Professional Python Volume 2: Advanced

Usharani Bhimavarapu and Jude D. Hemanth

Learning Advanced Python from Open Source Projects

Rongpeng Li

Foundations of Data Science with Python

John Mark Shea

Data Mining with Python: Theory, Applications, and Case Studies

Di Wu

A Simple Introduction to Python

Stephen Lynch

Introduction to Python: with Applications in Optimization, Image and Video Processing, and Machine Learning

David Baez-Lopez and David Alfredo Báez Villegas

Tidy Finance with Python

Christoph Frey, Christoph Scheuch, Stefan Voigt and Patrick Weiss

Introduction to Quantitative Social Science with Python

WeiQi Zhang and Dmitry Zinoviev

For more information about this series please visit: <https://www.crcpress.com/Chapman--HallCRC/book-series/PYTH>

Introduction to Quantitative Social Science with Python

Weiqi Zhang and Dmitry Zinoviev



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Designed cover image: © Shutterstock, Stock Photo ID 2362078849, Photo Contributor NicoElNino

First edition published 2025

by CRC Press

2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press

4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2025 Weiqi Zhang and Dmitry Zinoviev

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Zhang, Weiqi, 1983- author. | Zinoviev, Dmitry, author.

Title: Introduction to quantitative social science with Python / Weiqi Zhang and Dmitry Zinoviev.

Description: First edition. | Boca Raton, FL : CRC Press, 2025. | Series: Chapman & Hall/CRC the Python series | Includes bibliographical references and index.

Identifiers: LCCN 2024019625 (print) | LCCN 2024019626 (ebook) | ISBN 9781032354606 (hardback) | ISBN 9781032356594 (paperback) | ISBN 9781003327899 (ebook)

Subjects: LCSH: Social sciences--Research--Methodology. | Python (Computer program language) | Quantitative research.

Classification: LCC H62 .Z485 2025 (print) | LCC H62 (ebook) | DDC 300.72--dc23/eng/20240729

LC record available at <https://lcn.loc.gov/2024019625>

LC ebook record available at <https://lcn.loc.gov/2024019626>

ISBN: 978-1-032-35460-6 (hbk)

ISBN: 978-1-032-35659-4 (pbk)

ISBN: 978-1-003-32789-9 (ebk)

DOI: [10.1201/9781003327899](https://doi.org/10.1201/9781003327899)

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

*To my beloved wife, Juan Ling, and exceptional son,
George Zhang, who endured my countless hours of
writing. Without your love and the occasional reality
check, this journey would have been much longer.
Thank you for being my anchors. And to my students,
for whom this work is intended, may it guide and
inspire you.*

Weiqi Zhang

*To my wife, Anna, and my children, Eugenia and
Roman, whose love and care made this book possible.
To my friends Dmitry and Tatyana Cherevik, and
Pelin Bicen, who wholeheartedly supported me in
writing the book. And to the students at Suffolk
University, who have inspired me every day for the
last two years.*

Dmitry Zinoviev



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Contents

Preface	xv
Contributors	xvii
List of Figures	xix
List of Tables	xxiii
I “Executive Track”	1
1 Introduction to Data Analysis in Social Science	6
1.1 The Role of Data Analysis in Social Science Research	7
1.1.1 Identify a Research Question	9
1.1.2 Literature Review	9
1.1.3 Developing a Hypothesis	10
1.1.4 Testing the Hypothesis	11
1.1.5 Drawing Conclusions	17
1.2 What’s Next?	18
1.3 Discussion Questions	18
2 Data Collection and Cleaning	19
2.1 Data Collection: Sources and Repositories	19
2.1.1 Web Scraping	21
2.1.2 Data Storage and Exchange Formats	21
2.2 Data Organization	23
2.3 Data Preprocessing: Refining Raw Data for Analysis	24
2.3.1 Levels of Measurement	25
2.3.2 Data Types in Python	28
2.3.3 Data Issue 1: Mismatching Data Types	29
2.3.4 Data Issue 2: Missing Values	29
2.3.5 Data Issue 3: Outliers	30
2.4 What’s Next?	31
2.5 Discussion Questions	32

3	Descriptive and Exploratory Analysis	34
3.1	Statistics in Descriptive Analysis	34
3.1.1	Visualize Frequency Distribution	36
3.1.2	Measures of Central Tendency	41
3.1.3	Distribution Skewness and the Choice of Central Ten- dency Metrics	42
3.1.4	Variability	45
3.2	Statistics in Exploratory Analysis	47
3.3	What's Next?	50
3.4	Discussion Questions	51
4	Causality and Hypothesis Testing	52
4.1	Randomized Controlled Experiments	53
4.1.1	Limitations of Randomized Controlled Experiments	54
4.2	Observational Methods	54
4.2.1	Solutions to Confounders	56
4.3	Hypothesis and Hypothesis Testing	56
4.3.1	Step 1: Establish the Null Hypothesis	57
4.3.2	Step 2: Choosing a Testing Approach	59
4.3.3	Step 3: Selecting a Test Method, Calculate the Test Statistic, and Find the p-value	60
4.3.4	Some Common Test Methods	63
4.4	The Non-parametric Approach	70
4.5	What's Next	73
4.6	Discussion Questions	74
5	Linear Regression Analysis	75
5.1	Simple Linear Regression	76
5.1.1	The Ordinary Least Squares (OLS) Method	77
5.1.2	Example: Impact of GDP on Ladder Score	82
5.1.3	Assumptions of the OLS simple regression	83
5.1.4	Impact of Outliers on Regression Results	86
5.2	Linear Multiple Regression	87
5.2.1	Include Ordinal and Nominal Variables in Regression	90
5.2.2	Include an Interaction Term in Regression	92
5.3	What's Next	99
5.4	Discussion Questions	100
6	Classification	101
6.1	Modeling Dependent Variables with Two Categories	102
6.1.1	Example: Impact of Perceptions of Corruption on Bi- nary Income Group	103
6.1.2	Evaluating Goodness-of-Fit	105
6.1.3	Assumptions of Logistic Regression Analysis	106
6.2	Multinomial Logistic Regression	107

6.3	Ordered Logistic Regression	111
6.3.1	The Parallel Regression Assumption	112
6.4	Non-Parametric Approach: K-Nearest Neighbors Algorithm	113
6.4.1	The KNN Algorithm	115
6.4.2	Processes of Performing KNN Classification	116
6.4.3	Example: Estimating Income Group Classification	122
6.4.4	The Receiver Operating Characteristic (ROC) Curve and Area Under ROC Curve (AUC)	123
6.5	Stereotyping in Classification Models	124
6.6	What's Next	126
6.7	Discussion Questions	126
7	Complex Network Analysis	127
7.1	General Definitions	127
7.1.1	Regular and Complex Networks	128
7.1.2	Network Types and Building Blocks	129
7.2	Forms of Network Representation	132
7.3	Centralities and Densities	134
7.3.1	Closeness Centrality	134
7.3.2	Degree Centrality	135
7.3.3	Betweenness Centrality	136
7.3.4	Eigenvector Centrality	137
7.3.5	Global and Local Density	137
7.4	Network Structure Extraction	138
7.4.1	Cliques	138
7.4.2	Core-Periphery Analysis	139
7.4.3	Community Detection	140
7.5	Bipartite Networks	142
7.6	SNA as a Statistical Tool	144
7.7	What's Next	144
7.8	Discussion Questions	145
8	Text As Data	147
8.1	Text Processing Pipeline	147
8.1.1	Denoising	148
8.1.2	Language Detection	148
8.1.3	Text Corpora	149
8.1.4	Tokenization	150
8.1.5	Part-of-Speech Tagging	152
8.1.6	Stop Word Removal	152
8.1.7	Stemming and Lemmatization	153
8.2	Frequency Analysis	155
8.3	Summarization	156
8.3.1	Extractive Summarization	156
8.3.2	Abstractive Summarization	157

8.4	Named Entities Recognition	158
8.5	Sentiment Analysis	160
8.6	Topic Discovery	161
8.7	Discussion Questions	163
II	“Technical Track”	165
9	Python Programming Fundamentals	169
9.1	Programming Workflow	169
9.1.1	Integrated Development Environments	170
9.1.2	Program Execution	171
9.2	Variables, Data Types, Operations, and Operators	172
9.2.1	Variables	172
9.2.2	Numbers	173
9.2.3	Strings	174
9.2.4	Data Types	181
9.2.5	Reading and Displaying Data	183
9.3	Modules	185
9.3.1	Installing Extra Modules	186
9.4	What’s Next	186
9.5	Programming Exercises	187
10	Data Collection and Cleaning	188
10.1	Containers	188
10.1.1	Lists and Tuples	189
10.1.2	Dictionaries	192
10.2	Working with Files	194
10.2.1	Fundamentals of Text File Manipulation	194
10.2.2	Pickling	196
10.3	Pandas	197
10.3.1	Importing Data	197
10.3.2	Accessing Rows and Columns	199
10.3.3	Data Types and Levels	203
10.3.4	Row and Column Manipulation	205
10.3.5	Filtering	207
10.3.6	Handling Missing Values	208
10.4	Combining Dataframes	210
10.4.1	Merging and Joining	211
10.4.2	Merging Partially Matching Dataframes	212
10.4.3	Pivoting and Stacking	213
10.5	What’s Next	215
10.6	Programming Exercises	215

11 Condition Checking and Descriptive and Exploratory Analysis	217
11.1 Condition Checking and Decision-Making	217
11.1.1 Checking Membership	218
11.2 Handling Programming Errors	219
11.2.1 Preemptive Checking (Pessimistic Approach)	220
11.2.2 Reactive Recovery (Optimistic Approach)	220
11.3 Counting, Frequency Distribution, and Visualization	222
11.3.1 Counting with Counter	222
11.3.2 Counting with Pandas	223
11.3.3 Histograms and Bar Charts	224
11.4 Descriptive Statistics Functions	226
11.4.1 Data Imputation Revisited	227
11.5 Identifying and Interpreting Outliers	229
11.6 Utilizing Correlation to Analyze Variable Relationships	229
11.7 What's Next	232
11.8 Programming Exercises	232
12 Loops and Hypothesis Testing	234
12.1 Repeating Operations with Loops	234
12.1.1 "While" Loops	234
12.1.2 "For" Loops	236
12.1.3 List Comprehension	237
12.2 Parametric Hypothesis Testing	239
12.2.1 Shapiro–Wilk Test	239
12.2.2 Z-Test	240
12.2.3 T-Test	241
12.2.4 Analysis of Variance (ANOVA) Test	242
12.2.5 Chi-Squared Test	244
12.3 Non-Parametric Hypothesis Testing via Bootstrapping	245
12.4 Automating Boring Tasks	247
12.5 What's Next	248
12.6 Programming Exercises	248
13 User-Defined Functions and Regression Analysis	249
13.1 Writing User-Defined Functions	249
13.1.1 Choose the Function Name	250
13.1.2 Define the Interface	251
13.1.3 Implement the Function Body	252
13.1.4 Invoking a User-Defined Function	253
13.1.5 Document the Function Design	255
13.2 Linear Regression	256
13.2.1 Building Linear Regression Using Patsy	257
13.2.2 Using the Fitted Model for Predictions and Visualization	260

13.2.3	Conducting Linear Multiple Regression using Patsy	260
13.2.4	Incorporating Categorical Variables	262
13.2.5	Incorporating Interaction Terms	263
13.3	Visualizing Interaction Effects	264
13.4	What's Next	266
13.5	Programming Exercises	267
14	Generators and Classification	268
14.1	Generators	268
14.1.1	Built-In Generator Functions	269
14.1.2	Generator Expressions	271
14.2	Classification via Logistic Regression	272
14.2.1	Binary Logistic Regression	272
14.2.2	Multinomial Logistic Regression	273
14.2.3	Ordered Logistic Regression	275
14.3	K-Nearest Neighbors Classification	276
14.3.1	Normalization	277
14.3.2	Cross-Validation	278
14.4	Receiver Operating Characteristic Curve	280
14.5	What's Next	281
14.6	Programming Exercises	281
15	More Generators and Network Analysis	282
15.1	Generators, Revisited	282
15.2	NetworkX and Graph Creation	283
15.2.1	Creating a Graph	284
15.2.2	Accessing Nodes, Edges, and Their Attributes	287
15.3	Other Network Representations	288
15.4	Exploring Network Structure	289
15.4.1	Connected Components and Subgraphs	289
15.4.2	Projecting Bipartite Graphs	291
15.4.3	Cliques, Cores, and Communities	292
15.4.4	Density and Clustering Coefficient	294
15.5	Calculating Centralities	294
15.6	Network Visualization	295
15.7	What's Next	297
15.8	Programming Exercises	297
16	Sets. Text as Data	298
16.1	Sets	298
16.1.1	Set Properties and Operations	298
16.1.2	Uses of Sets	300
16.2	HTML and BeautifulSoup	301
16.3	Natural Language Toolkit (NLTK)	303
16.3.1	Tokenization and Language Detection	303

16.3.2 Part-of-Speech Tagging	304
16.3.3 Stop Word Elimination	305
16.3.4 Stemming and Lemmatization	306
16.4 Extractive Summarization	307
16.5 Named Entity Recognition	309
16.6 Sentiment Analysis	310
16.7 Topic Modeling	311
16.8 What's Next	313
16.9 Programming Exercises	314
Conclusion	315
A Solutions to Select Exercises	317
Bibliography	321
General Index	323



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Preface

In the spring of 2022, the idea for this book began to take shape as we recognized the limitations of existing proprietary analytical software. While R language thrived in academia, Python emerged as the dominant language for data science in the private sector. We saw an opportunity to introduce a free, powerful, and popular analytical tool for social science students, preparing them for future professional endeavors.

The primary goal of this book is to introduce Python as a powerful tool to readers and equip them with a conceptual understanding of quantitative analysis and practical skills aligned with modern data-driven research demands. Beyond establishing a solid foundation in quantitative analysis methodologies, we delve into practical analytical skills for addressing common tasks, including regression, classification, social network analysis, and text analysis.

Another goal of the book is to bridge social sciences and data science. Weiqi is a quantitative political scientist who has published a few quantitative social science studies and taught statistics and data analysis courses in Python. Dmitry is a computer science professor and data scientist who has authored many Data Science and Python books and collaborated with scholars from various disciplines. Together, we seek to contribute to the advancement of quantitative social science.

The structure of the book caters to readers with diverse technical backgrounds. A unique feature of the book is its inclusion of two tracks: the “executive track” and the “technical track.” The executive track focuses on conceptual aspects and methodologies of data analysis. Unlike traditional data analysis textbooks that often focus on formulas, our approach prioritizes clarity and understanding over complex mathematical formulas. We avoid presenting mathematical formulas in the chapters to make the content accessible to non-technical readers, allowing them to grasp the conceptual foundations without being hindered by complex mathematical notation. In contrast, the “technical track” focuses on the practical application of Python for the analyses discussed in the executive track.

The dual-track structure offers flexibility to readers, allowing them to tailor their reading experience based on their needs and familiarity with the subject matter. Those inclined to understand conceptual aspects first can progress through the executive track, followed by corresponding chapters in the technical track. Alternatively, readers interested in the conceptual aspects alone can

stay on the executive track, while experienced data analysts eager to explore Python could delve directly into the technical track. This adaptable structure ensures that readers can optimize their learning journey according to their preferences and requirements.

Weiqi Zhang and Dmitry Zinoviev
April 2024

Contributors

Weiqi authored [Chapters 1 to 6](#), and Dmitry authored the rest.



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>



List of Figures

1.1	Scientific research life cycle.	8
2.1	Variable categories and data types.	28
2.2	Distribution of GDP per capita in the world in 2021.	31
2.3	Distribution of log GDP per capita in the world in 2021.	32
3.1	Bar chart.	37
3.2	Bar chart (sorted).	37
3.3	Histogram of 2021 ladder score.	38
3.4	Histogram of ladder score (ten bins).	39
3.5	Histogram of ladder score (three bins).	39
3.6	Pie chart: relative frequency of regions.	40
3.7	Pie chart: relative frequency of ladder score ranges.	41
3.8	Symmetrical distribution ($N(0,1)$).	43
3.9	Left skewed distribution ($\Gamma(2,1)$).	44
3.10	Right skewed distribution ($\Gamma(2,1)$).	45
3.11	Multimodal distribution (two superimposed normal distributions).	46
3.12	Standard deviation.	47
3.13	Correlations between ladder score and other variables.	48
3.14	Correlations of a non-linear relationship.	49
3.15	Different associations with the same correlation coefficients.	50
4.1	A confounder in a spurious relationship.	55
4.2	Hypothesis testing process.	57
4.3	Normal distribution.	62
4.4	Test statistic for the 5% threshold in two-tailed test vs. in left-tailed test.	64
4.5	Illustration of the central limit theorem.	71
4.6	Sampling distribution of differences in means.	73
5.1	Relationship between happiness and economic development.	77
5.2	Identify the best fitted linear model.	78
5.3	Linear model for inches and centimeters.	78
5.4	Residuals of linear regression.	79
5.5	Explanation for variation in the dependent variable.	81
5.6	Testing homoscedasticity assumption.	84

5.7	Heteroscedastic residuals.	85
5.8	Testing normality of residuals.	86
5.9	Homoscedasticity test (without outliers).	88
5.10	Normality test (without outliers).	89
5.11	Parallel slope assumption in the relationship between corruption and confidence in government.	94
5.12	Effect of interaction in the relationship between corruption and confidence in government.	95
5.13	Parallel slopes of regions.	97
5.14	Interaction between region and corruption.	98
6.1	Forcing an OLS model on a categorical dependent variable.	102
6.2	Impact of corruption perceptions on income groups.	104
6.3	Estimated probabilities for three regions based on ladder scores and log GDP per capita.	110
6.4	Estimated probabilities under the ordered logistic regression (upper) and multinomial logistic regression (lower).	114
6.5	Estimate the shape of the unknown new data.	115
6.6	Process of KNN classification.	118
6.7	ROC curves and estimation performances.	124
6.8	ROC curves of logistic and KNN models.	125
7.1	Zachary karate club network.	128
7.2	A form-content network of Turkish numerals.	130
7.3	Maximum cliques in Zachary karate club network.	139
7.4	The main core of Zachary karate club network.	140
7.5	Four detected communities in Zachary karate club network.	141
7.6	Unweighted bipartite Davis Southern women event network. Node color denotes the two parts.	142
7.7	Weighted projected Davis Southern women event network (the attendants view); cf. Figure 7.7 on p. 143.	143
9.1	Spyder IDE with an empty new file in the editor window and Python 3.6.10 prompt in the console window.	170
10.1	Horizontal (a) and vertical (b) concatenation.	211
11.1	Histogram illustrating the distribution of generosity.	225
11.2	Bar chart illustrating the distribution of countries by region (vertical).	227
11.3	Bar chart illustrating the distribution of countries by region (horizontal).	228
11.4	Negative affect in India: regular values (dots) and outliers (crosses). Threshold= $\pm 1.5\sigma$	230
11.5	Healthy life expectancy at birth <i>vs.</i> log GDP per capita in 2021. The correlation between the variables is 0.855.	232

15.1 The Correlates of War Formal Alliance dataset in the 21st century represented as a complex network. 287

15.2 Weighted projected Davis Southern women event network (the event view); cf. Figure 7.7 on p. 143. 296

16.1 Venn diagram illustrating set operations on words. 300

16.2 Positive and negative sentiment levels in *Macbeth*. 312



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>



List of Tables

2.1 Example of tabular data 22

2.2 Wide data structure 24

2.3 Long data structure 24

3.1 Absolute and relative frequency distributions 35

3.2 Absolute frequency of ladder scores (2021) 36

4.1 Writing a hypothesis based on research question 58

4.2 Observed and expected frequencies of region and income group 69

4.3 Contribution two chi-squared statistic 70

5.1 Regression results 82

5.2 Regression results (without outliers) 87

5.3 Correlation table to test multicollinearity 88

5.4 Multiple regression results 89

5.5 Results with categorical variables in regression 92

5.6 Regression results with interaction term between two continu-
ous variables 93

5.7 Regression results with interaction term between corruption
and region 96

6.1 Relationship between income groups and corruption perception 105

6.2 Relationship between regions and ladder score and log GDP
per capita 109

6.3 Relationship between income group and life expectancy, free-
dom, and generosity 112

6.4 Confusion matrix for label values in two categories 117

6.5 Confusion matrix for label values in multiple categories . . . 119

6.6 KNN classification report 122

7.1 Partial adjacency matrix of an undirected, unweighted network
without self-loops. Cf. Figure 7.1 on p. 128. 132

7.2 Partial adjacency matrix of a directed, unweighted network
with a self-loop (boxed). Cf. Figure 7.2 on p. 130. 133

7.3 Partial edge list (left) and its CSV representation (right) of a
directed, unweighted network. Cf. Figure 7.2. 134

7.4 Five nodes with the highest closeness, degree, betweenness, and
eigenvector centralities, sorted by the closeness centrality in
Zachary’s karate club network. 135

7.5 Logistic regression analysis: determining factors influencing
karate club members’ decision to stay with the officer or leave
with Mr. Hi 145

8.1 Lemmatization and stemming results for select English words 154

8.2 The most frequently used n-grams in the Declaration of Inde-
pendence 155

8.3 Five topics in “Alice’s Adventures in Wonderland” by Lewis
Carroll and their most frequently used words with frequencies
in 1/1000th. 162

9.1 Predicate string functions 178

9.2 Conversion string functions 178

9.3 Search string functions 178

9.4 Truth table for the logical operators 181

9.5 Notable modules in the Python standard library 185

10.1 File input/output functions 194

10.2 Dataframe accessors 201

11.1 Functions for calculating statistics of central tendency 228

13.1 Results: Ordinary Least Squares 259

15.1 Correlations between network measures in the karate club net-
work 295

Part I

“Executive Track”



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction

Traditional data analysis education in social science places significant emphasis on performing manual calculations and working with numerical statistics. However, this approach no longer aligns with the real-world demands of the modern job market for data analysts. In today's professional landscape, data analysts rarely engage in manual number crunching, as modern computers possess the computational power to handle these tasks efficiently. Instead, the emphasis lies in understanding the rationale behind specific calculations and algorithms (a precise, finite, and stepwise sequence of rigorous instructions), as well as how to instruct computers to execute them.

The conventional educational approach also fails to cater to the diverse needs of contemporary learners. Some students may be prepared to absorb both conceptual and technical information simultaneously, while others might prefer a more gradual approach, starting with the conceptual aspects of data analysis and deferring the technical aspects until later stages of their learning journey.

This book takes an innovative approach, departing from tradition, to introduce fundamental concepts, frameworks, and tools for conducting data analysis in social science research. Each topic is structured into two complementary tracks: the Executive Track and the Technical Track. The Executive Track is designed for readers who wish to focus on the conceptual foundations of data analysis without delving into the technical intricacies of Python programming. Simultaneously, the Technical Track is tailored to those who are eager to apply the concepts discussed in the Executive Track through hands-on Python coding.

The Executive Track and Technical Track are interconnected, allowing readers to smoothly transition between the two. Key concepts and methodologies from the Executive Track are directly linked to the corresponding technical aspects covered in the Technical Track, ensuring a seamless learning experience for those who wish to bridge the gap between theory and practical implementation.

This book's primary focus is on equipping readers to become contemporary quantitative social science researchers. Consequently, it emphasizes the conceptual frameworks of data analytics in the Executive Track and practical programming skills in the Technical Track, with less emphasis on complex mathematical formulas and manual calculations.

Why Python?

Numerous tools are available for data analysis, including Excel, SPSS, Stata, SAS, R, and Python. When selecting a data analysis tool, a critical consideration is cost. Proprietary software options like Excel, SPSS, Stata, and SAS often come with prohibitively expensive licensing fees. While colleges may offer students discounted access during their courses, it is rare for students to afford renewing these licenses after graduation. Upon leaving the academic environment, individuals who do not work for large organizations with the financial means to support such software typically lose access to these tools and, consequently, the skills they have developed.

In contrast, open-source tools like Python and R offer a distinct advantage in that they are freely available for use. The absence of licensing costs makes them accessible to a broader audience. Moreover, due to their low cost of use, robust online communities have formed around these tools, contributing to their development and fostering a culture of mutual support and knowledge sharing. This accessibility and collaborative environment empower users to continually enhance their data analysis skills, irrespective of their financial means or affiliation with a large organization.

Python is a versatile programming language, and it introduces a fundamental concept in programming: “Don’t Repeat Yourself” (DRY). This principle emphasizes the importance of avoiding redundant or repetitive work.

Proprietary software often offers a user-friendly experience with graphical user interfaces (GUIs) and “point-and-click” features, making it approachable for users. However, it can be characterized as “Write Every Time” (WET) because these point-and-click interfaces are typically designed for one-time use. Analysts using proprietary software find themselves compelled to redo procedures every time they need to execute a sequence of actions or perform recurring analyses. This approach can lead to inefficiency and consumes valuable time, especially in complex analyses.

In contrast, programming languages like Python and R empower analysts to design a generalized sequence of instructions for a computer once. These instructions can then be effortlessly applied to various datasets and scenarios, adhering to the DRY principle. This capability significantly enhances the efficiency and effectiveness of data analysis, as analysts can avoid repetitive manual tasks and focus on creating adaptable and reusable code for their analytical needs.

Python and R are two prominent programming languages frequently utilized for data analysis. Python is the preferred choice in this book because of its wide adoption among data analysts. According to surveys conducted by Kaggle, one of the largest data science communities, 87% of data scientists consistently used Python from 2018 to 2021. In contrast, R’s usage was reported at 38% in 2018, declining to 30% in 2021 [kag21].

Python enjoys an additional advantage in terms of its vast and active online community. For example, there are over two million Python-related questions on StackOverflow, a question-and-answer website for programmers, while there are fewer than half a million R-related questions. This extensive support network makes it easier for learners to find solutions to their coding issues and seek help when needed.

Furthermore, Python's English-like syntax makes it more accessible for beginners to learn and understand, even if they lack a technical background.

One of Python's standout features is its versatility as a general-purpose programming language. It extends beyond data analysis to enable automation of daily tasks, web development, computer vision technologies like facial recognition, financial technology (fintech), mobile application development, and even game development. Learning Python for data analysis can open doors to a plethora of other technical skills and opportunities. In essence, Python is a versatile tool that can serve multiple purposes, and learning it can be seen as a decision to gain proficiency in a range of valuable skills, rather than just one. It's a choice that can address many diverse needs and interests.

Introduction to Data Analysis in Social Science

In the realm of social science research, there has long been a misconception that it is primarily associated with non-quantitative methods. Some students pursuing social science majors may have even chosen this path to steer clear of quantitative courses, such as mathematics and statistics. However, it is crucial to acknowledge that social scientists have consistently striven to incorporate scientific, quantitative research methods akin to those employed in natural sciences. In fact, the term “social science” was coined under the premise that studying societies should mirror the methodologies of scientific inquiry [KK05].

The last half-century has witnessed a remarkable revolution, marked by the proliferation of computational and information technologies. This transformation has empowered researchers with access to robust hardware and sophisticated software, enabling them to effortlessly collect and share data related to human behavior online, as well as to perform advanced analyses across a myriad of datasets.

Contemporary social science studies predominantly embrace quantitative data analysis. This paradigm shift is exemplified by the authors of this book, who have harnessed machine learning, text analysis, and network analysis to scrutinize North Korean foreign policy by discerning shifts in how North Korean official news media reports on China over the past two decades. They have also applied regression analysis to explore the relationship between Chinese leaders’ educational backgrounds and their political careers spanning five decades. Additionally, they employed factor analysis to construct an index and employed regression techniques to assess how the amalgamation of political, social, and economic institutions in an economy can either facilitate or impede technological innovation.

In diverse social science disciplines like psychology, data analysis assumes various forms, such as the measurement of personality characteristics through personality tests. In fields like sociology, literature, and history, researchers can digitize images, characters, geolocation data, or sound files and subject them to quantitative analysis.

The applicability of quantitative social science research extends well beyond academic research. There is a surging demand for data-driven solutions to address social, economic, and political challenges. Furthermore, while statisticians and data scientists are proficient in data analysis, not all

possess the domain expertise necessary to decipher the nuances of social science research [Sha11].

In response to this demand, many graduate-level social science programs and an increasing number of undergraduate programs, particularly in the United States, have incorporated data analysis into their mandatory curriculum. Renowned institutions like Harvard, UC Berkeley, Carnegie Mellon, and Columbia, to name a few, offer specialized programs in quantitative social science.

This book has been crafted with the intent of making data analysis more accessible to a broad readership, including those without a technical background. It aims to bridge the gap and equip individuals with the necessary skills to navigate the evolving landscape of data analysis in the realm of social science.

1.1 The Role of Data Analysis in Social Science Research

In contemporary social science research, a prevailing trend is the integration of knowledge and skills from multiple disciplines. This interdisciplinary approach requires a synthesis of expertise in social science to discern what to analyze, statistics to ascertain which analytical methods to employ and how to interpret the results, and computer science to harness computational technology for data collection, analysis, and data visualization.

It is crucial to emphasize that data analysis is but one component within the broader spectrum of the social science research process. A proverbial phrase, “To a man with a hammer, everything looks like a nail,” aptly illustrates a common pitfall encountered by novice data analysts. This pitfall involves an excessive fixation on data analysis methods while neglecting the essential aspects of other research components.

Data analysis skills can be likened to tools, much like a hammer, which enable us to perform actions. However, it is important to recognize that the tool itself does not prescribe its suitability for a given task. The onus falls on researchers themselves to determine which type of analysis or tool is appropriate for their research, as there is no universal, one-size-fits-all method.

Before immersing ourselves in the realms of data, Python programming, and statistical techniques, it is imperative to comprehend the role that data analysis plays within the context of social science research.

Social science research typically consists of five steps:

1. Identify the research question.
2. Review the literature critically and develop a hypothesis.
3. Test the hypothesis.
4. Report results.
5. Draw a conclusion.

Data analysis plays a pivotal role in steps 3 and 4. Each step contains more caveats. [Figure 1.1](#) provides an overview of the research life cycle. The following sections will delve deeper into each of these steps.

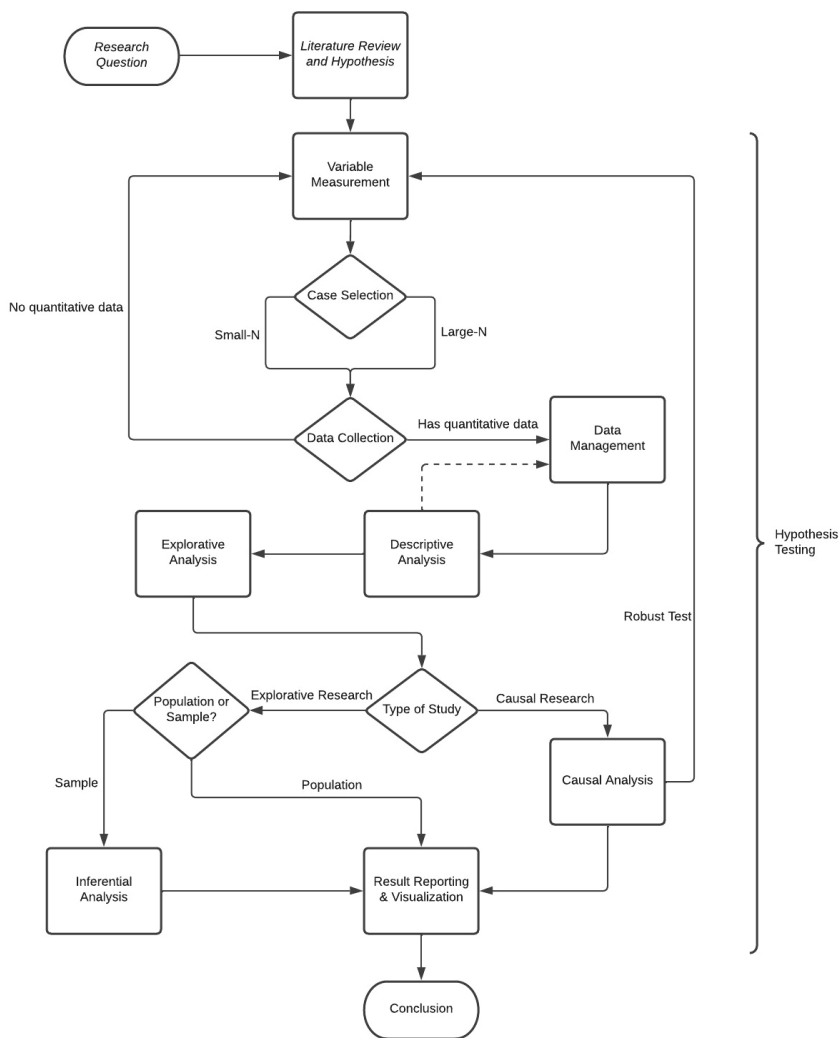


FIGURE 1.1
Scientific research life cycle.

1.1.1 Identify a Research Question

At the heart of any research project lies the research question, a pivotal component that sets the course for the entire endeavor. It is essential to recognize that not every question qualifies as a research question; some are driven merely by curiosity. When formulating a research question, it is crucial to ask a fundamental query: why should others care about our research? A compelling research question is one that others find worthwhile addressing.

There are two primary categories of research questions: exploratory and causal. Exploratory research questions are designed to address descriptive aspects of a topic. For example, they might inquire about changes in the number of gun owners in each U.S. state over the past 50 years. The response to such questions is descriptive, as it elucidates changes but does not delve into the underlying reasons. Exploratory questions are often employed to “explore” areas of study that have not been previously investigated, potentially opening doors to new areas of research.

In contrast, causal research questions seek to answer the “why” behind a particular phenomenon. Instead of asking how gun ownership has changed, a causal question would be, “Why has the number of gun owners in each state changed in a particular manner over the past 50 years?” Responding to this question entails examining multiple factors and discerning their interplay.

An effective causal research question often revolves around a puzzle, a phenomenon that does not conform to standard patterns and cannot be readily explained by existing theories. For instance, a perplexing question in comparative politics is, “Given the historical failures and collapses of most communist regimes and the theoretical explanations for the inherent weaknesses of communist systems, why has China achieved political stability and sustained economic growth for several decades?” Solving such puzzles not only enriches our understanding of existing patterns but also leads to the development of new theories.

For term papers, theses, or academic journal publications, research questions should ideally be narrowly defined. A well-focused research question serves to maintain our concentration and ensures that the research project remains manageable. We can narrow the scope of their questions by imposing limitations, whether in terms of time or geographic coverage. Similar to completing a complex jigsaw puzzle with smaller, manageable pieces, conducting several narrowly defined research projects on a topic can pave the way for addressing broader research questions, such as Daron Acemoglu and James Robinson’s extensive work in their book *Why Nations Fail*, wherein they analyze the historical evolution of political, social, and economic changes in numerous major economies throughout human history.

1.1.2 Literature Review

Literature review constitutes the second vital component of a research project. It serves as an amalgamation of what other scholars and researchers have

previously articulated or discovered regarding the research question or any topic closely related to it. Beyond mere compilation, a literature review necessitates a critical evaluation of diverse viewpoints and theories, requiring an insightful assessment of their strengths and weaknesses. This evaluation hinges on the explanatory power of these theories. A theory with greater explanatory power can illuminate a broader spectrum of cases, rendering it a more robust theoretical framework.

To enhance the explanatory power of theories, we must focus on identifying and addressing the weaknesses inherent in existing theories, often referred to as “theoretical gaps.” Bridging these gaps is tantamount to reinforcing the theory’s foundation. It is crucial to recognize that all theories are abstractions of reality and, as such, are inherently prone to weaknesses.

When scrutinizing the literature, an effective approach is to question the validity of theories by asking, “Could the theory be wrong?” We should investigate several areas to uncover these theoretical gaps, including:

1. **Assumption and Logic:** Are the assumptions of a theory sound, and does the logic hold up under scrutiny?
2. **Limitations:** Are there any limitations imposed by the theory, such as constraints on time or geographic coverage? Are these assumptions sound, and does the logic hold up under scrutiny?
3. **New Factors:** Investigate whether the theory takes into account emerging or previously unconsidered factors that could influence the phenomenon under study. Failing to incorporate these factors may result in an incomplete or outdated theoretical framework.
4. **Analytical Methods:** Are there potential shortcomings in the methodology used to test the theory’s hypotheses?

By critically assessing these dimensions, researchers can uncover and address theoretical gaps, ultimately strengthening the theoretical foundation and explanatory power of their research. The literature review, therefore, serves as a springboard for constructing a more comprehensive understanding of the research question and identifying avenues for expanding our knowledge within the realm of social science.

1.1.3 Developing a Hypothesis

Hypotheses are the cornerstone of any research endeavor. Following a comprehensive literature review, we formulate a hypothesis, a concept that plays a pivotal role in addressing theoretical gaps. A hypothesis represents an idea that we aim to substantiate, and it is accompanied by a clear articulation of expected observations based on this idea. In essence, a hypothesis should outline what one should anticipate observing if it proves to be correct.

Consider the scenario of explaining China’s robust societal and economic performance under a communist government. One might propose a hypothesis that suggests that, despite the communist political system, the Chinese

government has adeptly responded to evolving challenges by meticulously tailoring leadership recruitment standards. The goal is to select leaders equipped to tackle specific societal problems. If this hypothesis holds true, one would expect to observe intentional promotions of different types or styles of leaders within the Chinese Communist Party as the country's needs evolve.

Moreover, since conventional wisdom often distinguishes between communist or authoritarian governments and democratic ones in the realm of political development, this hypothesis possesses the potential not only to shed light on the successes and failures of communist governments but also to enrich our understanding of democracies. This expansion of explanatory power enhances the relevance and applicability of existing theories.

A hypothesis is more than a mere assumption; it is a driving force behind research, guiding the investigative process, and serving as a framework for the interpretation of empirical observations. A well-formulated hypothesis lays the foundation for rigorous analysis and contributes to the advancement of social science knowledge.

1.1.4 Testing the Hypothesis

Scientifically testing a hypothesis entails transforming it into a formal model with variables. At the core of every hypothesis lies the concept of variables. Variables help us structure and quantify the phenomena we aim to investigate.

Each hypothesis must include at least one dependent variable. This variable represents the central focus of the research, and we endeavor to explain its changes. In exploratory research, typically, one dependent variable is examined. For instance, when exploring shifts in gun ownership across U.S. states, "gun ownership" serves as the dependent variable.

Causal research, on the other hand, necessitates the inclusion of both a dependent variable and at least one independent variable. The independent variable is the factor that we believe has the potential to cause changes in the dependent variable. In our previous example, the dependent variable remains "gun ownership," while one independent variable could be "economic growth." This hypothesis posits that a stronger economy leads to increased disposable income, which, in turn, translates into higher gun ownership. However, it is important to note that this relationship is merely a hypothesis and requires empirical testing for validation.

In more intricate hypotheses (theoretical models), there may also be control variables. Control variables are introduced to differentiate between control groups and treatment groups, serving to eliminate extraneous factors that could confound the results. Additionally, intervening variables might be incorporated into the model when the influence of an independent variable on the dependent variable is contingent upon other factors. These nuances and their practical applications will be explored in greater detail in subsequent chapters.

The terminology used in social science disciplines may vary in different contexts or across academic fields. For instance, what we refer to as variables

are sometimes known as factors or characteristics. The dependent variable may be called by different names, such as “response variable,” “predicted variable,” “output,” “label,” or “target.” Likewise, the independent variable may be referred to as the “explanatory variable,” “feature,” “predictor variable,” or “input.” These variances in terminology highlight the interdisciplinary nature of research and the necessity for us to adapt to the conventions of specific fields or contexts.

Measuring Variables

Following the construction of a theoretical model, we encounter the pivotal task of determining how to measure the variables within that model. In contrast to the precision commonly found in natural science research, the measurement process in social science research can be considerably more complex due to several factors.

First and foremost, the nature of social science subjects presents inherent challenges. Unlike natural sciences, where variables like distance and temperature can be precisely and universally measured, social science grapples with variables and phenomena that evade precise measurement. Concepts such as the quality of democracy, an individual’s political orientation (e.g., liberal or conservative), emotions, intentions, culture, and trust are inherently nuanced and often resist direct quantification. Furthermore, these variables are not always directly comparable. For example, in a survey, two individuals who both “agree” with a statement may not necessarily agree with each other on the same statement, reflecting the inherent complexity of social phenomena.

Consequently, when a variable cannot be directly observed or measured, we must seek proxies, variables that exhibit a strong or a similar pattern of change, with the variable of interest. For instance, economic competitiveness is a concept that is challenging to measure directly. To address this, we might employ the market share of the largest ten companies in the economy as a proxy. The underlying assumption is that in a competitive economy, larger companies tend to have smaller market shares, as opposed to a monopolistic one where larger companies might dominate. To enhance the accuracy of a proxy, we might turn to factor analysis, a technique that employs a group of variables to estimate the unmeasurable one. It is important to note that proxies require a causal relationship and are not infallible. Studies using proxies typically involve robust testing to bolster the reliability of findings, a topic we will delve into further in [Chapter 4](#) on p. 52.

At this stage, we are in the planning phase of data collection, setting the groundwork for subsequent empirical investigations.

Selecting Cases

Case selection involves two critical facets: how to select cases and how many cases to select. The “how” primarily depends on the accessibility to the subject population, which is determined by the research question. In instances where

we can access the entire subject population (all subjects of interest), selecting all subjects is the ideal approach. For example, if the research question focuses on student academic performance in a specific class, the population consists of all students in that class, and we may aim to include all of them in the study. However, more often than not, we lack access to the entire population and must work with samples, a subset of the population, for analysis.

Random sampling, which entails the random selection of cases, stands as the gold standard for case selection in scientific research. Sampling and the challenges encountered by social science researchers will be elaborated upon in [Chapter 2](#) on p. 19.

One approach to categorizing research is based on sample size, which encompasses two types: large-N studies and small-N studies. Large-N studies feature extensive sample sizes, typically with hundreds or more cases, while small-N studies involve limited sample sizes, typically fewer than 30 cases. The sample size can be constrained by resource limitations or influenced by the nature of the study. For instance, research exploring rare phenomena, such as successful communist countries, minority students in classrooms, or specific psychological illnesses, often necessitates small sample sizes.

From the perspective of quantitative hypothesis testing, large-N studies are better suited for intricate and complex analyses (as explored in [Chapter 12](#) on p. 234). In contrast, small-N studies present challenges in hypothesis testing due to a lack of cases to control differences and similarities among subjects, making them more suitable for qualitative research involving descriptive data analysis. It is vital to recognize that a small sample size does not necessarily constitute a hindrance. Researchers can design case selection strategies, such as the “most-similar research design” and the “most-different research design,” to bolster hypothesis testing. Readers interested in these approaches can delve into the literature on qualitative research design for further insights.

A quantitative solution to address small sample sizes is to transform a small-N study into a large-N study by altering the unit of analysis. The unit of analysis represents the primary subject of focus for analysis and data collection. For example, a study on public policy could scrutinize the impact of a policy across 50 U.S. states, where the state serves as the unit of analysis, yielding a sample size of 50. By shifting the focus to the impact of the policy on individuals and households, researchers can significantly increase the sample size to tens of thousands or even millions of observations.

Historical analysis studies can adopt similar conversions. For instance, a study analyzing the impact of North Korea-China economic relations on North Korea’s foreign policy from 1997 to 2017 can face sample size challenges when using a “year” as the unit of analysis, yielding only 20 observations (one year of data per observation). To address this limitation, we can opt to use “month” as the unit of analysis, increasing the sample size to 240 (12 months per year multiplied by 20 years), thereby providing a sufficient dataset for a large-N study.

Collecting Data

Data collection and management represent pivotal stages in the preparation of data for analysis, often referred to as “extract-transform-load” or ETL in data science. “Extract” pertains to data collection, “transform” concerns data management, and “load” refers to importing data for subsequent analysis. While these topics will be briefly addressed in this chapter as an integral part of the research process, they will be explored in greater depth in [Chapter 2](#).

The information revolution has significantly enhanced the accessibility of data on a wide range of topics. A vast array of information is now publicly available on the internet for downloading. Some organizations, such as X (formerly Twitter), provide application programming interfaces (APIs) that enable users to directly search for and retrieve data from organization databases. Data obtained from these sources can be easily imported into Python for processing. In cases where data, like news articles, cannot be readily downloaded but is available on web pages, we can construct web scraping functions in Python to extract information from websites. However, web scraping may present legal challenges. Before initiating web scraping, we must review the website’s terms of service to determine whether it permits the extraction and utilization of its data without explicit permission, a topic explored further in [Chapter 2](#).

It is not uncommon for researchers, particularly those conducting experiments or survey research, to collect their own data. In situations where quantitative data for a variable cannot be obtained, we must consider alternative methods for measurement and subsequently embark on the process outlined earlier until all variables are equipped with quantitative data. This iterative process ensures that data is collected and transformed into a format conducive to rigorous analysis.

Managing Data

Ideally, collected data should be ready for analysis, often referred to as “clean data.” In practice, however, data almost always require some level of management before analysis. Data management encompasses the organization of data structure (transformation) and data cleaning. Here are some key considerations in data management:

Merging Data: When downloaded data is stored in different sources, it needs to be integrated into a single dataset for analysis. This is typically accomplished through data merging techniques.

Structuring Data: The same dataset can be structured in various ways to address different research questions. We must appropriately structure the data according to the specific research question at hand.

Data Types: Data in variables can sometimes be saved in incorrect data types. Numeric values must be stored as numbers for analysis. For example, the value “4” can be mistakenly saved as a string, treating it as a character like the letter “a,” or as an integer (a number without fractions). Sometimes,

non-numeric characters can be mixed with numeric values. This forces Python to consider the values as strings instead of numeric values. For example, the value of one thousand dollars can be stored as “\$1,000”. Even though the meaning of “\$1,000” is clear to human eyes, the “\$” and “,” tell Python that they are not numeric values but characters. We need to ensure data types align with the analysis requirements.

Data Conversion: We may need to convert data to suit research needs. For instance, individual annual income can be converted from the actual dollar amount to thousands of dollars or categorized into groups, such as upper, middle, or lower economic classes.

Missing Values: Missing values can present challenges in data analysis. Observations with missing values are typically excluded from the analysis, which could be problematic if there are too many missing values. Strategies for addressing missing values are more of an art than science and will be explored in detail in [Chapter 2](#).

Data Errors: Errors can occur in datasets, especially when original data are collected manually. For example, “100” might be mistakenly saved as “1000,” or text may be saved in lowercase for one observation and in uppercase for another. We must also check for duplicate observations in the data, which can arise when merging or stacking datasets. Unexpected duplicates should be identified and removed.

Effective data management is a pivotal step in data analytics. As the saying goes in data science, “garbage in, garbage out.” The quality of data significantly influences the quality of analysis and the resulting insights. It is not uncommon for researchers to invest more time in data management than in data analysis, underscoring its critical role in the research process.

Describing and Exploring Data

Quantitative analysis commences with descriptive analysis, which serves two primary objectives. The first is to offer an overview of the data to be analyzed, addressing the question of “what happened.” Some common metrics scrutinized in descriptive analysis encompass the dataset’s size, including the number of variables and observations, averages, ranges, proportions, and the variability of variables. A second objective is to verify that the data is prepared for analysis. Certain analytical methods are more sensitive to data quality than others. For example, regression analysis results can be influenced by outliers, which are observations exhibiting extreme values that deviate from the data’s typical pattern. Detecting outliers during descriptive analysis, especially if regression analysis is planned, necessitates additional data management to address these data anomalies.

It is essential to note that the presence of outliers does not inherently denote an unfavorable situation. Outliers, when observed, can trigger curiosity and exploration, potentially leading to novel research questions. Indeed,

some of the most compelling research inquiries stem from anomalies or puzzles within the data.

Exploratory analysis extends beyond descriptive analysis, delving into the examination of relationships among variables in the data. A prevalent metric employed to detect associations in exploratory analysis is correlation. It is crucial to underscore that exploratory analysis focuses on identifying associations among variables rather than establishing causation.

Data management, descriptive analysis, and exploratory analysis will be explored in depth in [Chapter 3](#) on p. 34, providing a comprehensive understanding of the intricacies involved in these stages of data analysis.

Inferential and Causal Analysis

The next stage of data analysis diverges into two paths contingent on the type of research being conducted. For exploratory research, the approach varies depending on whether we work with data pertaining to the entire population (population parameters) or a subset of that population (sample statistics). If we have data about the entire population, then we possess all the necessary information. However, when dealing with a sample, we need to employ inferential analysis to leverage sample statistics and estimate unknown population parameters. A quintessential example is public opinion polling in U.S. presidential elections, where pollsters gather the opinions of approximately one thousand individuals (sample) to gauge the sentiments of the entire nation (population). [Chapter 4](#) delves into the fundamentals of these two types of analysis, anchored in the domain of probability. [Chapter 12](#) is dedicated to elucidating how probability theory is used to evaluate sample statistics.

Causal research takes a distinctive trajectory, employing causal analysis, typically using regression analysis in social science research to unearth causal relationships between dependent and independent variables. It endeavors to address questions surrounding “why something happened” and can be used to respond to hypothetical queries like “what might occur if variable X changes by Y.” [Chapter 5](#) on p. 75 is a focal point for causal analysis and introduces two prevalent regression models employed in social science research. Additionally, text analysis ([Chapter 8](#) on p. 147), which converts qualitative text into quantitative data, and network analysis, which studies relationships and interactions among entities in a network ([Chapter 7](#)), are invaluable tools applicable to both exploratory and causal analysis.

In cases where the results of causal analysis support the hypothesis, we should consider conducting a robust test. A robust test assesses the resilience of the results to changes in the data used. We deploy different research methods or data measurements to validate whether the analysis results remain consistent. When the initial analysis and the robust test converge on the same conclusion, we can assert the results with greater confidence.

However, if the hypothesis is not substantiated by the analysis results or if the results do not withstand the robust test, we may explore alternative

research designs or analytical methods, collect additional data, introduce more variables into the analysis, or simply acknowledge that the hypothesis lacks support based on the available evidence. In the latter scenario, we should not be disheartened by the “unsuccessful” research, as the discovery that something is incorrect also contributes to the advancement of human knowledge.

Reporting and Visualizing Results

Reporting findings is a crucial aspect of the research process, ensuring that they are communicated effectively to the readers. A comprehensive analysis report typically comprises three fundamental components.

First, we should present key quantitative information derived from the analysis, often in a concise tabular format. Predictive analytics may also be utilized to extend beyond the immediate analysis results.

Second, we must provide a textual interpretation of the quantitative information. It is worth noting that in social science, despite the terminology “predictive analysis,” most researchers prefer to use terms like “forecast” or “estimate” rather than “predict.” This practice parallels how we refer to “weather forecasts” rather than “weather predictions.”

Third, the old adage “a picture is worth a thousand words” holds true, especially in data analysis. We should always try to employ data visualization techniques to represent the analysis results and forecasts, enhancing the accessibility of the research to a broad audience ([Chapter 3](#)).

1.1.5 Drawing Conclusions

In the final stage of the research process, we conclude by providing a concise summary of several key aspects:

Research Question: Revisit the research question, highlighting its significance and the broader context in which it is situated.

Contribution: Articulate the unique contribution the study makes to the field of research, emphasizing the value it adds to existing knowledge.

Assumptions and Hypothesis: Acknowledge and discuss the assumptions underpinning the analysis and the alternative hypothesis.

Research Method: Briefly outline the research methods employed in the study.

Findings: If the analysis supports the hypothesis, these findings are presented. However, in cases where the hypothesis is not supported, this should be objectively reported.

Limitations: It is customary to critically review the limitations of the study, which may pertain to assumptions, research methods, or data quality. We should ask ourselves, “Could I be wrong?” This section provides a balanced assessment of the study’s boundaries.

Future Directions: We can suggest potential avenues for further research to enhance the study’s overall quality and impact.

1.2 What's Next?

The next steps in your journey through the world of data analysis and social science research depend on your preferences and objectives. Here are two distinct paths you can choose from.

Data Management and Collection (Conceptual Focus): If you want to delve deeper into the conceptual aspects and are not yet ready to dive into coding, proceed to [Chapter 2](#). In this chapter, you will explore essential topics related to data management, including common data file formats, best practices for organizing and structuring data, and strategies for dealing with data quality issues (such as missing values).

This chapter will equip you with a strong foundation in data management and collection, which is vital for successful data analysis.

Introduction to Python (Hands-On): If you are eager to roll up your sleeves and start working with Python, you can jump ahead to [Chapter 9](#) on p. 169. In this chapter, you will embark on a hands-on journey through the basics of the Python programming language. You will learn how to create and assign values to variables, essential mathematical and logical operations in Python, the built-in data types of Python, how to use functions to perform tasks, and techniques for reading data and displaying results.

By choosing this path, you will acquire practical Python skills that are essential for data analysis and research in the social sciences.

1.3 Discussion Questions

1. What role does data analytics play in social science research? How do you think data analysis can benefit social science researchers, even those without a technical background?
2. Which is more important to data analysis: appropriate measured, clean dataset or complex algorithm or analytical model?
3. Which type of study is better suited for data analysis: small-N studies or large-N studies? Why?
4. What is the difference between descriptive, inferential, and causal analysis?
5. What steps are involved in the data analysis process, from identifying research questions to drawing conclusions?

Data Collection and Cleaning

Data quality is the cornerstone of effective data analysis. Think of conducting analysis as constructing a house, and data as the essential building blocks. Just as the quality of bricks affects the durability and strength of a building, data quality significantly impacts the reliability and accuracy of your analysis. In the world of data science, there is a common adage: “garbage in, garbage out.” It succinctly emphasizes that no matter how advanced your analytical methods may be, poor data quality will inevitably lead to poor results. Data quality, in turn, hinges on meticulous data preparation, which encompasses data extraction, transformation, and loading.

2.1 Data Collection: Sources and Repositories

In this digital age, we have the privilege of harnessing an abundant data ecosystem that transcends the need to collect original data. Instead, many researchers opt for data amassed and curated by third-party entities such as major organizations or fellow researchers. The Information Revolution has paved the way for easier access to a wealth of data, most of which is generously shared on online platforms. The pursuit of reliable, high-quality data is foundational to any meaningful analysis, and this section elaborates on several prominent data sources.

Government Websites: Government websites represent a goldmine for acquiring credible data about specific societies or communities. Virtually all governments systematically collect data about various facets of their societies and subsequently disseminate it via their official websites. As a prime illustration, the United States (US) government shares a plethora of datasets on www.data.gov. Likewise, the US Federal Reserve furnishes comprehensive data pertaining to the financial industry and the US economy on its website. These official government repositories are indispensable sources for a broad spectrum of analyses.

International Organizations: For studies of global scope or international nature, international organizations, such as the United Nations and the World Bank, offer a treasure trove of information. Their websites serve as veritable reservoirs of data, often supplementing or diversifying the range of data available on national government websites. These organizations amass comprehensive datasets that enrich cross-border and international research endeavors.

Professional Organizations and Associations: Proficiently catered to specialized domains, professional organizations and associations emerge as instrumental data sources in particular areas. These entities rigorously curate and maintain datasets, ensuring their accuracy and relevance. A case in point is the National Association of Realtors, an esteemed repository of data focused on the real estate industry in the United States. These niche sources are of immense value, providing data and insights unattainable elsewhere.

Corporate Data Repositories: Acknowledging the escalating value of data, corporations have taken proactive measures to aggregate data from diverse sources, both public and private, and make them accessible to the public. Exemplifying this is Google's Public Data Explorer, which meticulously collects data from various governments and organizations, enhancing researchers' data arsenal. Additionally, tools like Dataset Search allow us to conduct keyword-based searches across diverse datasets. Amazon Web Services (AWS) Open Data program and Microsoft's Azure Open Datasets are further exemplars, presenting data on a broad spectrum of subjects. Prominent data sources like Statista, Kaggle, and Gapminder are favored by data enthusiasts for their diverse and meticulously curated datasets.

Academic Repositories: Some universities, research institutes, and individual researchers host and share datasets. Harvard Dataverse serves as a commendable repository of datasets used in academic research across various disciplines. These scholarly vaults are valuable resources for accessing and utilizing datasets that underpin scholarly publications and research endeavors.

Application Programming Interface: Going beyond traditional data downloads, the online landscape also offers APIs. APIs streamline the process of data retrieval by enabling users to programmatically request data. Instead of manually navigating websites and selecting data for download, APIs empower computer programs to autonomously retrieve information. One compelling use case is Twitter API, which facilitates the extraction of detailed tweet information directly within a Python program. RapidAPI, a popular platform, simplifies the search and access to a diverse array of APIs by bridging the gap between users and API suppliers. This new era of data access has elevated efficiency and broadened possibilities for data-driven research endeavors.

2.1.1 Web Scraping

Web scraping is the art of automated data extraction from websites and web pages. It encompasses a spectrum of techniques, from simple copy-paste to complex automated scripts, designed to retrieve structured data from the Internet. Web scraping is a specialized field and falls beyond the scope of this book, but resources like Ryan Mitchell's *Web Scraping with Python* offer comprehensive guidance for those eager to delve into this subject.

However, the ability to scrape data from websites should be exercised judiciously. Note that not all data on the web are meant for unrestricted access. If a website or data owner does not offer a direct download option or an API, it often implies that the data is safeguarded by copyright or restricted for other reasons. In such cases, it is essential to respect the terms of service of the website and, if necessary, seek permission from the data owner before proceeding with web scraping.

Additionally, when contemplating web scraping, we must consider privacy concerns and the nature of the data. Some information might be sensitive or contain private details, while other data could be part of public records. We should distinguish between publicly available data, which can often be obtained directly from government sources, and private or copyrighted information. For instance, in the United States, the Freedom of Information Act mandates federal and state governments to disclose certain previously unreleased information upon request. This legal framework facilitates access to public records while safeguarding the privacy and copyright of sensitive information.

2.1.2 Data Storage and Exchange Formats

Sharing data on the internet entails using various data formats to facilitate data extraction and compatibility. The choice of data format depends on the type of data being stored or exchanged. Here, we explore some common data formats used for data storage and sharing.

XLSX (Excel Spreadsheet XML): Excel utilizes this format to store data in tabular structures. Data in XLSX files are arranged in tables with rows and columns, making it ideal for organizing structured data.

CSV (Comma-Separated Values): This format represents tabular data as plain text with values separated by commas in rows. CSV files are memory-efficient and can be opened by numerous applications, including Excel. They serve as a standard for exchanging data between programs.

JSON (JavaScript Object Notation): Frequently used in conjunction with APIs, JSON is employed for data exchange between web application servers and clients. It features human-readable data presentation and supports various data types. In JSON, data is stored in pairs of names and values, and it can accommodate non-tabular data, such as network data.

TABLE 2.1
Example of tabular data

Name	Gender	Rank	GPA
Alice	Female	Sophomore	3.5
Bob	Male	Freshman	3.82
Chuck	Male	Senior	3

XML (eXtensible Markup Language): XML is a versatile data format suitable for data storage and exchange between programs. It shares readability and data type support with JSON but distinguishes itself by using tags to define elements and attributes to represent data.

Table 2.1 and Listings 2.1, 2.2, and 2.3 illustrate how tabular data is structured in CSV, JSON, and XML formats. Although data can be manually entered in these formats, they are typically generated or saved through computer programs. Chapter 10 on p. 188 delves into reading and saving data in these formats using Python Pandas library. Additionally, some data analysis programs like Stata, SAS, and R employ proprietary data formats, which can be converted to CSV or handled using Python libraries such as Pandas. This ensures compatibility and seamless data exchange across different platforms.

Listing 2.1
example.csv

```
Name,Gender,Rank,GPA
Alice,Female,Freshman,3.50
Bob,Male,Sophomore,3.82
Chuck,Male,Senior,3.00
```

Listing 2.2
example.json

```
[
  {
    "Name": "Alice",
    "Gender": "Female",
    "Rank": "Freshman",
    "GPA": 3.5
  },
  {
    "Rank": "Sophomore",
    "Name": "Bob",
    "Gender": "Male",
    "GPA": 3.82
  },
  {
    "GPA": 3,
```

```
        "Name": "Chuck",  
        "Rank": "Senior",  
        "Gender": "Male"  
    }  
]
```

Listing 2.3

example.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<Class>  
  <Student>  
    <Name>Alice</Name>  
    <Gender>Female</Gender>  
    <GPA>3.50</GPA>  
    <Rank>Freshman</Rank>  
  </Student>  
  <Student>  
    <Gender>Male</Gender>  
    <Rank>Sophomore</Rank>  
    <GPA>3.82</GPA>  
    <Name>Bob</Name>  
  </Student>  
  <Student>  
    <Rank>Senior</Rank>  
    <Gender>Male</Gender>  
    <GPA>3.00</GPA>  
    <Name>Chuck</Name>  
  </Student>  
</Class>
```

2.2 Data Organization

The organization of data has a substantial impact on the analysis process. Typically, in tabular data, each column corresponds to a variable, with each row holding information about all variables for a single observation. The initial row often contains variable names. Python's `Pandas` library, a popular tool for managing and analyzing structured data, uses dataframes as its primary data structure. Dataframes, a two-dimensional structure, arrange data into rows and columns.

Data can also be organized in wide or long structures. To illustrate, let's consider the World Happiness Report [HLS+22], which assesses global subjective well-being. The report employs people's opinions to compute Ladder scores that gauge happiness levels in various countries. Table 2.2 reports the

TABLE 2.2
Wide data structure

Countries	2020 Ladder	2021 Ladder	2020 GDP (logged)	2021 GDP (logged)
Canada	7.024905	6.939435	10.73422	10.77075
United States	7.028088	6.959088	11.00441	11.05517

Ladder scores and logged GDP values for the US and Canada in 2020 and 2021 (why the GDP values have log transformation will be discussed in 2.3.5 on p. 30).

In the wide format, the first row contains variable names, which incorporate time information. This format is reader-friendly and well-suited for visually demonstrating changes within one variable across multiple categories. However, its significant drawback is that as new variables are introduced, each one results in two additional columns, potentially making the dataset too unwieldy.

Conversely, the long data structure treats years as values of the “Year” variable and records data for different variables and years in separate rows (Table 2.3). Although less intuitive for human readers, the long structure accommodates more variables within the table, adding only one column per new variable. As we delve deeper into upcoming chapters, it becomes evident that this structure is more conducive to data analysis, offering greater flexibility and utility.

2.3 Data Preprocessing: Refining Raw Data for Analysis

Data preprocessing is a vital step in the data analysis pipeline. It involves converting raw data into a refined, consistent, and well-structured format that is conducive to analysis. Despite the abundance of data on diverse subjects, a majority of datasets harbor imperfections necessitating preprocessing. Common issues include inappropriate data types, missing values, and outliers.

The significance of data preprocessing in the data analysis life cycle is underscored by the 80/20 rule prevalent in the data science industry. This rule

TABLE 2.3
Long data structure

Countries	Year	Ladder	GDP (logged)
Canada	2020	7.024905	10.73422
Canada	2021	6.939435	10.77075
United States	2020	11.00441	7.028088
United States	2021	11.05517	6.959088

posits that analysts typically allocate 80 percent of their time to preprocessing data and the remaining 20 percent to the actual analysis. It emphasizes the critical role of data preparation in facilitating meaningful and accurate insights from the subsequent analysis.

2.3.1 Levels of Measurement

Statistical data fall into distinct categories known as levels of measurement, which elucidate how variables are assessed. These data must be converted into Python data types for subsequent processing, as explained in [Section 9.2](#) on p. 172. To clarify the distinction between the two concepts: data types dictate the operations Python can perform, while the level of measurement indicates which operations are logically applicable.

Social or behavioral phenomena must be measured in some way to be analyzed. There are four levels of measurement:

- Nominal (also called qualitative or factor)
- Ordinal
- Interval
- Continuous

The selection of the level of measurement for a variable can be straightforward in certain instances and more nuanced, dependent on the research design, in others. The choice holds importance as it determines the types of analyses and inferences that can be drawn from the data.

Nominal. Nominal variables pertain to characteristics or categories without inherent numeric significance, rendering them qualitative in nature. Examples include gender, culture, and ethnicity. In quantitative analysis, numeric values are assigned to these categories to facilitate computation. For instance, gender may be encoded as follows: male = 0 and female = 1. In the case of ethnicity, the assignment might look like this: white = 1, Black or African American = 2, Hispanic/Latino = 3, Asian = 4, American Indian/Alaska Native = 5, Native Hawaiian and Other Pacific Islanders = 6, and Multiracial = 7.

It is important to note that these numeric assignments lack any inherent order or hierarchy. They serve as placeholders and are essentially arbitrary. Consequently, values of nominal variables should not be subjected to mathematical operations. The numeric representation merely aids in data processing and analysis without implying any quantitative meaning or superiority among the categories.

Ordinal. Ordinal variables are employed to measure categories that exhibit a clear order, but the intervals between them are not uniform or precisely quantifiable. Examples of ordinal variables include people's preferences on a

Likert scale (ranging from “strongly disagree” to “strongly agree”), educational levels, social classes (e.g., lower class, middle class, upper class), and product quality ratings (ranging from “good” to “bad” or 1 to 5 stars).

While ordinal variables, like nominal variables, describe non-numeric characteristics, they also incorporate information about the relative order within their categories. This crucial feature distinguishes ordinal variables from nominal ones. Therefore, when choosing the appropriate level of measurement for non-numeric variables with distinct categories, nominal variables are suitable for those without an inherent order, while ordinal variables are fitting for those with an observable order.

Numeric values assigned to ordinal variables should align with the hierarchy or order among category levels. For instance, a common way to measure preferences using a Likert scale is to assign values as follows: “strongly disagree” = 1, “disagree” = 2, “neither disagree nor agree” = 3, “agree” = 4, and “strongly agree” = 5. The order can be reversed without issue, as long as the values signify the relationship between the levels. The specific range of values, such as 1 to 5, is arbitrary and can be adjusted as needed.

Another example is the U.S. census, which provides information about individuals’ educational backgrounds. The educational categories encompass options like “less than 9th grade,” “9th to 12th grade, no diploma,” “high school graduate,” “some college, no degree,” “Associate degree,” and “Bachelor’s degree or more” (with four sub-categories). An educational background variable in the census data qualifies as an ordinal variable because more years of education are generally considered “higher” in order than fewer years of education. Researchers can assign numbers from 1 to 6 to these categories, with 1 indicating the lowest educational level and 6 representing the highest.

It is important to recognize that ordinal variables are limited in their ability to indicate whether the intervals between their levels are uniform or quantifiable. In the context of educational backgrounds, if we assign a value of 1 to “less than 9th grade” and 6 to “Bachelor’s degree or more,” we merely acknowledge that the latter group is more educated than the former. However, we cannot conclude that the latter group is six times more educated than the former group.

Similarly, in the case of measuring people’s preferences, for example, we can determine that respondents who choose “strongly disagree” are less supportive than those who choose “disagree.” Likewise, we understand that those who choose “disagree” are less supportive than those who select “neither disagree nor agree,” and so on. However, we cannot ascertain whether the difference in negativity between the former pair is equivalent to that between the latter pair. Put differently, we cannot determine whether individuals who choose “disagree” (coded as 2) are twice as supportive as those who choose “strongly disagree” (coded as 1), and whether those who choose “agree” (coded as 4) are four times more supportive.

This limitation highlights the challenge of interpreting ordinal variables in terms of the magnitude of differences between their categories.

Nominal and ordinal variables fall under the umbrella of categorical variables, which organize data into discrete groups or classes, offering a fundamental way to understand and classify information within a dataset.

Interval. Interval variables exhibit all the characteristics of ordinal variables, but they possess an additional quality: the intervals between their values are equal and meaningful. In other words, the numerical difference between any two points on the scale is consistent.

A classic example of an interval variable is temperature. The difference between 10 degrees and 20 degrees is the same as the difference between 20 degrees and 30 degrees. Moreover, the difference between 20 degrees and 40 degrees is twice as large as the difference between 10 degrees and 20 degrees. This uniformity allows for meaningful mathematical operations involving intervals.

However, it is essential to note that interval variables lack an absolute zero point. An absolute zero point signifies the absence of the subject of interest. For instance, zero degrees in temperature does not indicate the absence of temperature but rather a low temperature. Similar examples include coordinates and time. Consequently, when dealing with interval variables, calculating ratios may not be meaningful when zero is involved.

Continuous. Continuous variables share similarities with interval variables in that they measure precise differences between observations, making them suitable for quantifying non-whole (fractional) values. These variables are ideal for describing attributes like age, weight, and income.

One key distinction between continuous and interval variables is that continuous variables can encompass an unbroken spectrum of values within a defined range, while interval variables feature equidistant intervals. Additionally, continuous variables have the advantage of an absolute zero point, which allows for meaningful ratio calculations. In this context, a value of zero signifies the complete absence of the attribute under consideration. For example, a weight of zero indicates no weight, and zero income implies no earnings. Even an age of 0 signifies that someone is a newborn. This attribute makes continuous variables especially useful for calculating meaningful ratios when zero is part of the equation.

Precision of Levels of Measurement

The four levels of measurement vary in terms of their precision and flexibility. Continuous variables offer the highest level of precision and flexibility, as they can represent any value within a specified range. Interval variables come next, with predefined intervals that limit their flexibility. Ordinal variables lose some precision but retain the ordering of categories. Nominal variables, on the other hand, are the least precise, essentially serving as labels that provide no information about the relationships between observations.

In practice, precision is not always better. Researchers often convert high-precision variables into lower-precision categories. For example, actual income

values might be grouped into income levels (e.g., low, middle, high), or ages could be categorized as young, mid-aged, and senior. The reason for this conversion is that excessive precision can introduce noise and obscure commonalities among individuals within the same category. It is important to note that this conversion is a one-way street; variables can only be converted from high precision to low precision, not the other way around.

Mathematical operations intended for continuous variables sometimes are applied to ordinal variables, such as calculating the average of respondents’ responses using numeric Likert scales. Product ratings on e-commerce websites are generated in a similar manner. While these operations may not be technically appropriate because ordinal variables lack precision, they are sometimes used when precise information is not readily available.

2.3.2 Data Types in Python

In Python, a data type refers to a collection of values that share the same properties and support similar operations (see [Section 9.2](#) on p. 172 for examples of the operations). The built-in data types in Python include strings, integers, floating-point numbers (commonly referred to as “floats”), and Boolean. Python also allows third-party modules to introduce additional data types to enable new properties and operations. These data types serve as the foundation for various operations in Python, providing a basis for more advanced data manipulations and calculations.

[Figure 2.1](#) illustrates the relationships between levels of measurement and data types in Python. Strings, which are collections of characters like “John”

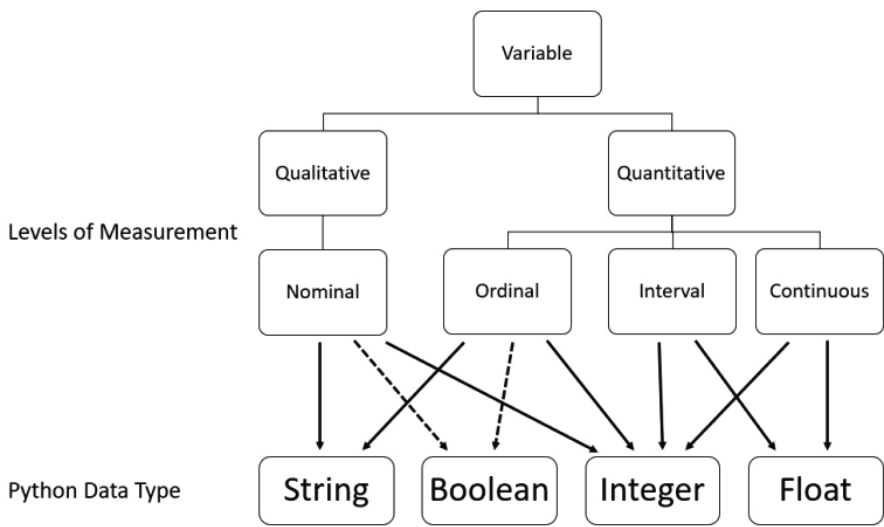


FIGURE 2.1
Variable categories and data types.

and “Jill,” can effectively represent nominal and ordinal variables, making them suitable for descriptive and exploratory analysis. Integer numbers encompass all positive and negative whole numbers, including zero, and can be employed to quantify nominal, ordinal, and interval variables such as ethnicity and preferences. In the case of variables with substantial values, like population size or national economy, they can be treated as continuous variables and stored as integers. Floating-point data is capable of approximating any real number with sufficient accuracy and is typically used for continuous and interval variables. For further details on the approximation of floating-point numbers, refer to [Chapter 9](#) on p. 169.

Python includes a unique data type known as Boolean, which exclusively consists of two values: True and False. It is applicable for nominal and ordinal variables with two categories. Moreover, True and False can be equated to 1 and 0, enabling them to interact with arithmetic operations. This characteristic is particularly valuable for data aggregation tasks, including determining the proportion of observations satisfying specific conditions.

2.3.3 Data Issue 1: Mismatching Data Types

Data types that do not align properly with the data values can lead to errors in Python. One common problem is when numbers are saved as string data type. Sometimes, spotting this issue is easy, as the values will contain string characters. For example, instead of storing one thousand dollars as “1000,” some data may use formats like “\$1,000” or “1000 USD,” where symbols like the dollar sign, comma, or “USD” imply the use of the string data type.

In other cases, identifying the problem can be more challenging. For instance, if “1000” is saved as a string, it may appear as an integer on the computer monitor. Data type mismatches can be troublesome because strings can be concatenated in Python. Adding two “1000” strings results in “10001000,” while adding two numeric 1000 values yields 2000. Furthermore, in Python, you can multiply a string by a number too, causing the string to repeat that many times. So, multiplying “1000” as a string by 2 gives “10001000.”

To prevent unintended data errors, we must thoroughly review data types for all variables in their dataset and ensure they are stored correctly before conducting any analysis.

2.3.4 Data Issue 2: Missing Values

Another common data issue is dealing with missing values. A missing value indicates the absence of information for a particular observation. Missing values are commonly represented as empty cells in CSV and XML files, NULL values in JSON, and None or NaN (“not a number”) values in Python. It is important to note that even if a value appears missing on your computer screen, it may still be present. For instance, an empty string is not typically considered a missing value but rather a valid string.

Missing values can pose challenges in data analysis because they are typically ignored in data operations. In descriptive or exploratory analysis, excessive missing values can lead to unreliable results. For more advanced analyses like machine learning, missing values become more problematic. If, for example, you want to perform regression analysis on a dataset with a dozen variables and one hundred observations, even a single missing value for one variable can lead to the entire observation's exclusion from the analysis. The more observations with missing values, the more exclusions from the analysis.

To address missing values, it is essential to minimize their presence in a dataset. The ideal approach is to collect more data, but this may not always be feasible. When the number of missing values is relatively small, they can be removed from the dataset.

In cases where the number of missing values is substantial and losing observations is not an option, data imputation can be considered. Data imputation involves replacing missing data with estimated values, aiming to make the best possible guess about the missing values. Several common imputation methods are used:

Mean, Median, or Mode Imputation: In this approach, missing values are replaced with the mean (average), median (middle value), or mode (most frequent value) of the variable, depending on the variable's type and distribution.

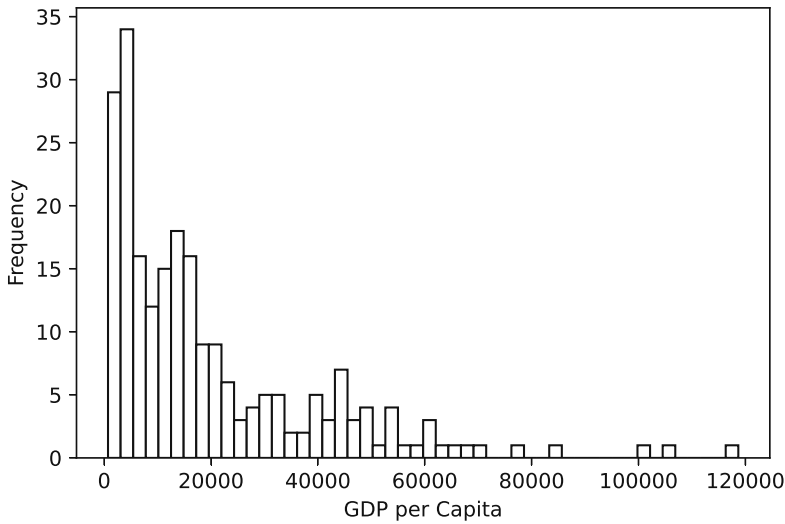
Nearest Neighbor Imputation: This method imputes missing values with values from similar observations. It is especially useful when there is a clear pattern in the data. For example, if historical economic data like national GDP has missing values due to various reasons, and there is a stable growth pattern, a missing value can be imputed using the values of nearby years.

Probabilistic Imputation: This approach involves fitting a statistical model, such as regression, to the data without missing values or similar observations. The model is then applied to estimate the most probable value for the missing values. For example, if a country's GDP has a stable annual growth rate, this rate can be used to estimate a missing value.

It is crucial to exercise caution when using data imputation. Imputed values are essentially made up, and they can introduce biases into the analysis. Relying on imputed data to identify patterns may lead to a self-fulfilling narrative. Therefore, data imputation should be considered a last resort, and any imputed values should be justified carefully.

2.3.5 Data Issue 3: Outliers

Outliers are observations whose values are significantly different from the rest and can impact data analysis. How to manage them depends on their causes. If outliers are due to errors, they should be corrected or deleted. In cases with outliers in a large sample, outright deletion is not advisable because extreme values could be found in a sufficiently large sample simply because of chance. Deleting these values can introduce bias.

**FIGURE 2.2**

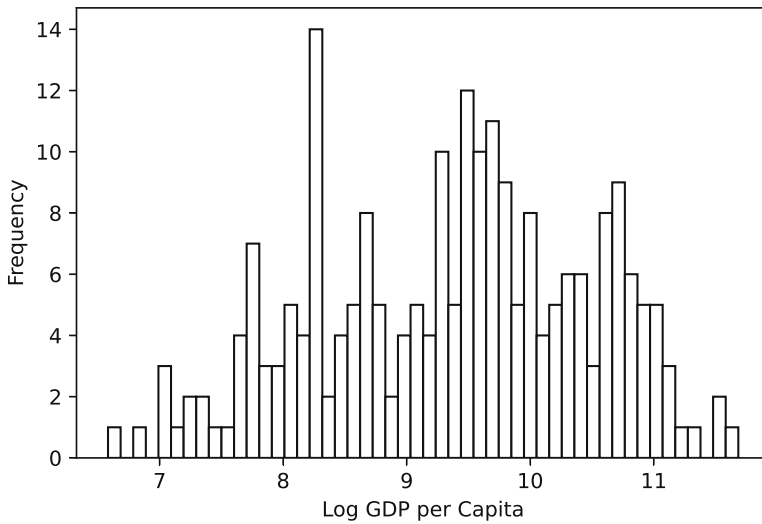
Distribution of GDP per capita in the world in 2021.

A better approach is to retain the outliers and apply data transformation techniques to make them less extreme compared to other data points. For instance, natural log transformation can compress the range of values and make the distribution more symmetric. This transformation is demonstrated in [Figures 2.2](#) and [2.3](#). The log-transformed distribution ([Figure 2.3](#)) displays a smaller range and a more symmetric shape compared to the original distribution ([Figure 2.2](#)).

The choice of how to manage outliers depends on the specific context and goals of the analysis, and we will explore outlier detection and management in [Chapter 3](#) on p. 34.

2.4 What's Next?

In [Chapter 3](#) on p. 34, you will dive into descriptive and exploratory data analysis. Descriptive analysis involves numerical summaries of variables or datasets, providing a foundational understanding of the data. Exploratory analysis aims to uncover preliminary patterns and associations in the data, often using visualizations. These techniques are essential for researchers to gain insights, refine research questions, and guide further analysis.

**FIGURE 2.3**

Distribution of log GDP per capita in the world in 2021.

[Chapter 10](#) on p. 188 will equip you with skills to:

1. Load data from various formats to Python.
2. Manage data structure and index of Pandas dataframe.
3. Evaluate and convert Python data types.
4. Perform vectorized operations to transform entire columns of data.
5. Extract and modify individual data cells.
6. Identify and handle missing values.

These tools will empower you to efficiently work with data, preparing you for more advanced analyses in your data science journey.

2.5 Discussion Questions

1. Why is data preprocessing important to data analysis?
2. What are the advantages and disadvantages of long and wide data structures?

3. Open a dataset of your choice, can you identify the appropriate variable types and levels of measurement for all variables?
4. What are some common methods for handling missing values in a dataset? When should you consider data imputation, and what are the potential pitfalls of imputing data?
5. What role does data transformation play in dealing with outliers?

Descriptive and Exploratory Analysis

After preparing a clean and structured dataset, we proceed with descriptive and exploratory data analysis. Descriptive analysis involves uncovering the fundamental characteristics of variables, providing insights that deepen our understanding of the data. On the other hand, exploratory analysis delves into uncovering relationships and associations between variables. Both forms of analysis serve as crucial building blocks for more advanced causal and inferential studies, often augmented with data visualization.

In this chapter, we will focus on the essential statistical methods commonly employed for the descriptive and exploratory analysis of structured data. You will also learn how to effectively communicate these findings through data visualization techniques.

3.1 Statistics in Descriptive Analysis

Descriptive analysis serves as the foundation for understanding the characteristics of your data and its variable values. It encompasses various key elements, starting with fundamental information about the data, such as the sample size (number of rows) and the number of variables (columns). Together, these details define the dimensions of your dataset, referred to as the dataframe in Python `Pandas` library.

Another crucial facet of descriptive analysis revolves around examining the values of variables. We often begin by investigating the frequency distribution of these variable values. The frequency distribution provides insights into the occurrence frequency of unique observations or values within a variable. To construct a frequency distribution, values within a variable are grouped into predefined categories.

By counting the number of observations in each category, researchers obtain absolute frequencies, which shed light on the distribution of values. This categorization process is relatively straightforward for nominal and ordinal variables, which encompass attributes like gender, country of origin, or satisfaction scale. As an example, consider the “Regional indicator” variable from the World Happiness Report, which classifies countries into cultural and economic regions. Absolute frequencies in this context reveal the number of

TABLE 3.1
Absolute and relative frequency distributions

Region	Absolute Frequency	Relative Frequency
East Asia and Pacific	12	11.43%
Europe and Central Asia	38	36.19%
Latin America and Caribbean	17	16.19%
Middle East and North Africa	10	9.52%
North America	2	1.9%
South Asia	5	4.76%
Sub-Saharan Africa	21	20%
Total	105	100%

countries sharing the same regional indicator value, aiding in understanding the distribution of countries across regions.

Besides absolute frequency, we may also find relative frequency to be a valuable statistic. Relative frequency provides insight into the occurrence of values concerning the total number of observations, offering a normalized perspective.

Table 3.1 illustrates both absolute and relative frequency distributions for the “Region” variable within the World Happiness Report’s 2021 dataset. In the second column, you can observe the absolute frequency for each region. For instance, the data set contains 12 countries (observations) classified under the “East Asia and Pacific” region. Given the total number of observations in 2021, which is 105 (accounting for missing values in some countries), the relative frequency for this region is 11.43% (12 out of 105).

It is worth noting that since these categories encompass all observations, the sum of relative frequencies always equals 100%.

Applying absolute and relative frequency analyses directly to continuous variables is not practical because continuous variables often have numerous distinct values, making it challenging to work with. For instance, consider the “Life Ladder” variable in the 2021 World Happiness Report, which ranges continuously from 0 to 10. Each unique value in this variable is essentially a distinct category since no two observations share the same Ladder score. The abundance of categories and the rarity of observations in each category make this approach impractical.

As an alternative, we can convert continuous variables, such as the Ladder score, annual income, or age, into categorized ranges, effectively transforming them into ordinal or nominal variables. It is crucial to ensure that these categories are both mutually exclusive and collectively exhaustive. In other words, each observation (value) in a variable must fit into precisely one category.

For instance, the “Life Ladder” variable could be divided into ten categories. The first category may encompass values “greater than or equal to 0 and smaller than 1” or denoted as “[0, 1),” while the second category could include values “greater than or equal to 1 and smaller than 2” or denoted as

TABLE 3.2
Absolute frequency of ladder scores (2021)

Range	Absolute Frequency	Relative Frequency
[0, 1)	0	0%
[1, 2)	0	0%
[2, 3)	2	1.68%
[3, 4)	7	5.88%
[4, 5)	24	20.17%
[5, 6)	35	29.41%
[6, 7)	40	33.61%
[7, 8)	11	9.24%
[8, 9)	0	0%
[9, 10)	0	0%

“[1, 2),” and so forth (we can determine our preferred categorization criteria). These categories should neither contain gaps nor overlap to ensure their mutual exclusivity and collective exhaustiveness.

Table 3.2 presents the absolute and relative frequency distributions for the Ladder scores following the categorization scheme. Notably, certain score categories, such as “[0, 1)” and “[1, 2),” have no observations associated with them. The category “[2, 3)” encompasses two observations, representing 1.68% of the total observations. This frequency distribution analysis helps us gain insights into the distribution of Ladder scores across different ranges.

3.1.1 Visualize Frequency Distribution

As the old saying goes, one picture is worth a thousand words. When it comes to frequency distribution, visual representations of data can be much more impactful than mere numbers. It is worth noting that data visualization techniques and tools are continually evolving, allowing for more sophisticated and insightful presentations of data. If you are interested in exploring further, you can apply the concepts and principles introduced in this chapter, coupled with your creativity, to create more complex and informative visualizations. How to generate data visualization in Python is covered in Section 11.3.3 on p. 224.

Absolute Frequency

To visualize the absolute frequency distribution of a categorical variable, such as the “Region” in Table 3.1, a bar chart is a useful tool. A bar chart typically includes a vertical axis representing frequency and one or more bars representing variable categories.

Figure 3.1 presents a bar chart based on the data in Table 3.1, with the bars arranged alphabetically. Additionally, the same bars can be organized based on frequencies, resulting in Figure 3.2. The sorted bar chart offers a clearer comparison of frequencies among categories. For instance, the

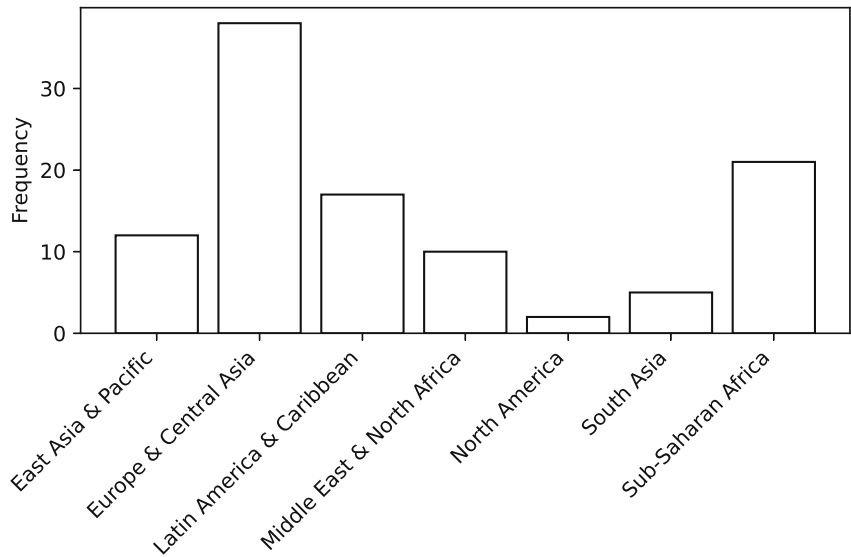


FIGURE 3.1
Bar chart.

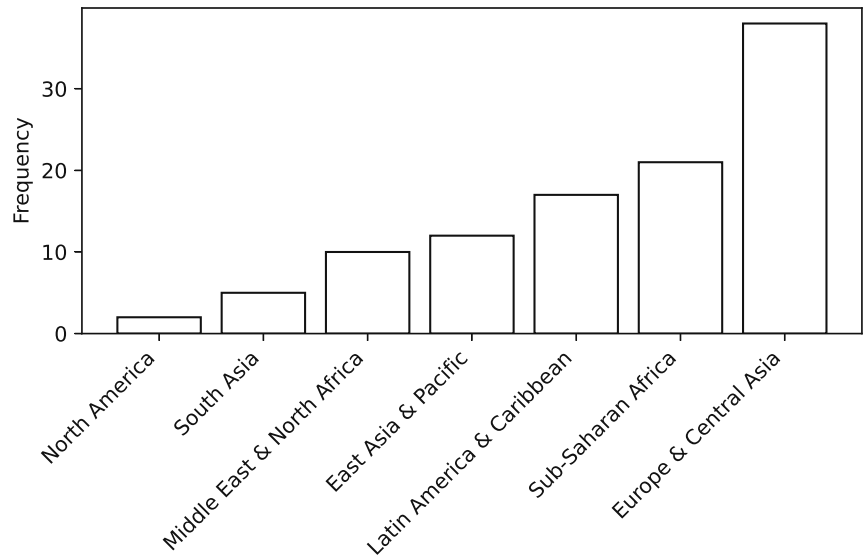


FIGURE 3.2
Bar chart (sorted).

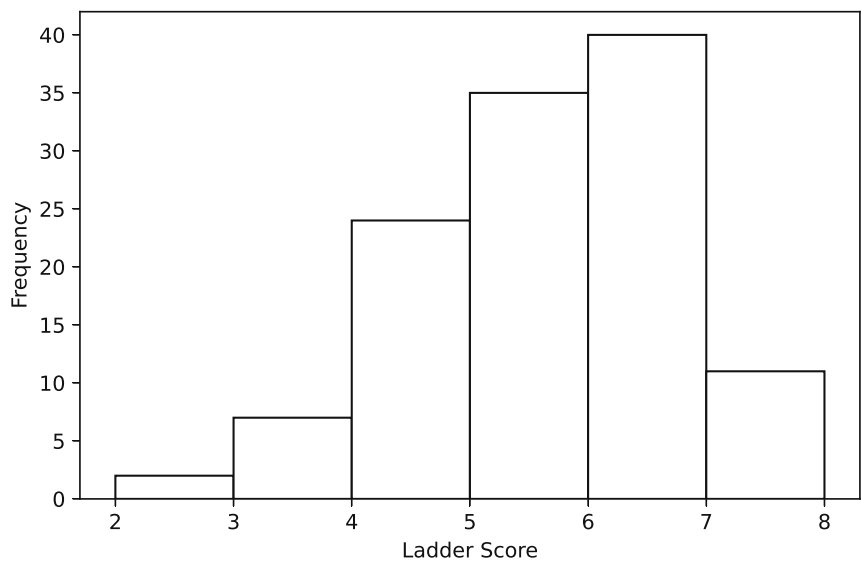


FIGURE 3.3
Histogram of 2021 ladder score.

difference between “East Asia and Pacific” and “Middle East and North Africa” is more apparent in [Figure 3.2](#), where it shows that the former has a slightly higher frequency than the latter.

Histograms are commonly used to visualize the distribution of continuous variables, like the Ladder Score in [Figure 3.3](#), which is based on the data in [Table 3.2](#). Histograms have a vertical axis representing absolute frequency and a horizontal axis representing a continuous variable. The data is divided into multiple bins, with the height of each bin indicating the frequency of observations within that range.

While histograms and bar charts may seem similar at first glance, they are fundamentally different. Bar charts are ideal for categorical variables, allowing the horizontal axis to be labeled arbitrarily with no specific numerical order. In contrast, histograms are tailored for continuous variables and demand a systematic numerical labeling in an ascending or descending order on the horizontal axis.

It is important to note that we have the discretion to choose bin widths of a histogram, which directly affect its number of bins and overall shape. This decision is more of an art than a science. For instance, [Figures 3.4](#) and [3.5](#) are based on the same data as [Figure 3.3](#) but with different bin widths. [Figure 3.4](#) employs ten bins with equal intervals, offering a more detailed representation of the data distribution. In contrast, [Figure 3.5](#) utilizes three bins with arbitrarily determined intervals, which conceals distribution details and could

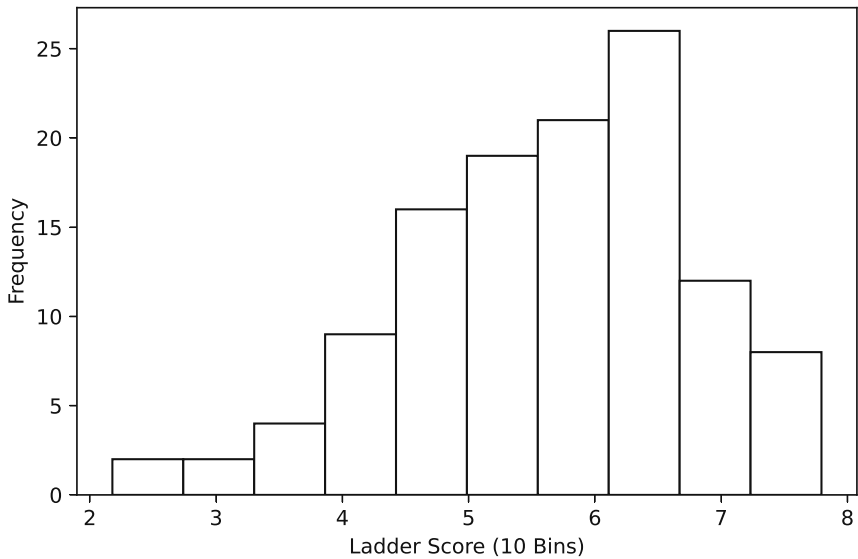


FIGURE 3.4
Histogram of ladder score (ten bins).

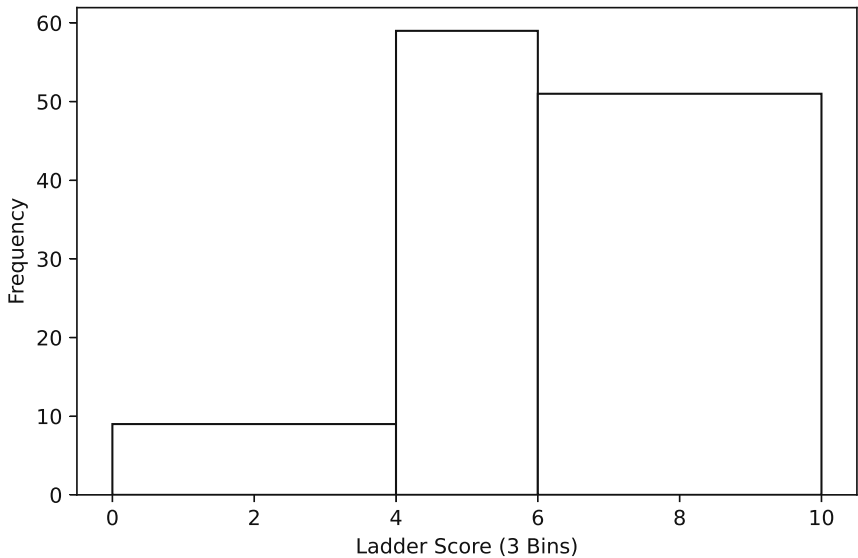


FIGURE 3.5
Histogram of ladder score (three bins).

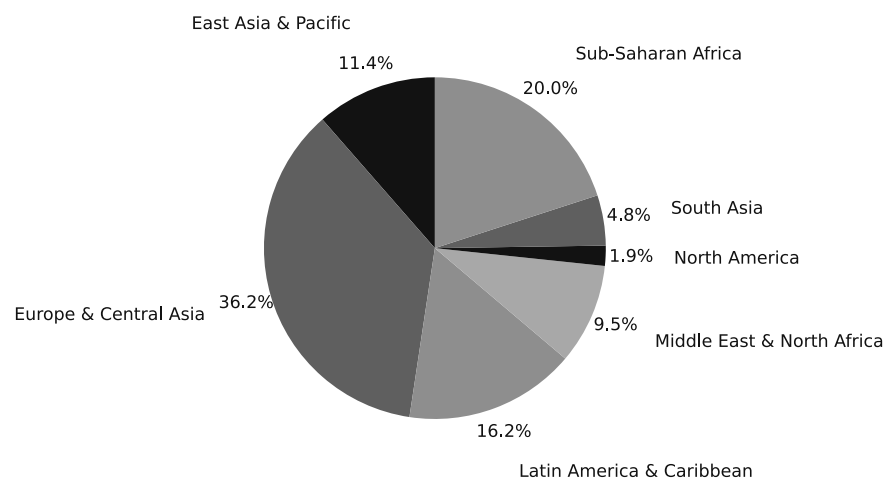


FIGURE 3.6
Pie chart: relative frequency of regions.

potentially mislead readers. Additionally, the latter histogram employs bins with varying widths, with some covering ranges without any observations. Therefore, careful consideration is required when selecting bin widths, as it can significantly impact data interpretation.

With that being said, it is not uncommon for researchers with domain expertise to establish categories with various intervals for analytical purposes. For instance, the US News categorizes economic classes for a family of three into lower income (annual income below \$52,000), middle income (\$52,200–\$156,600), and upper income (more than \$156,600) [Wal23], whereas the US Census Bureau employs seven intervals to create ten income levels (categories).

Relative Frequency

One key characteristic of relative frequency is that the total sum of relative frequencies for all categories in a dataset is always 100% or 1. A common way to depict relative frequencies is through a pie chart.

Figure 3.6 is based on the data in Table 3.1 and represents the relative frequencies of various regions in the 2021 World Happiness Report dataset. For instance, it reveals that countries in East Asia and Pacific constitute 11.43% of all the observations. Figure 3.7, on the other hand, illustrates the data from Table 3.2, emphasizing that observations with Ladder scores between 2 and 3 make up only 1.7% of the total, while those with scores between 5 and 6 account for 29.4%.

It is essential to note that pie charts have certain drawbacks and may not be the most suitable visualization tool. Pie charts excel when dealing with

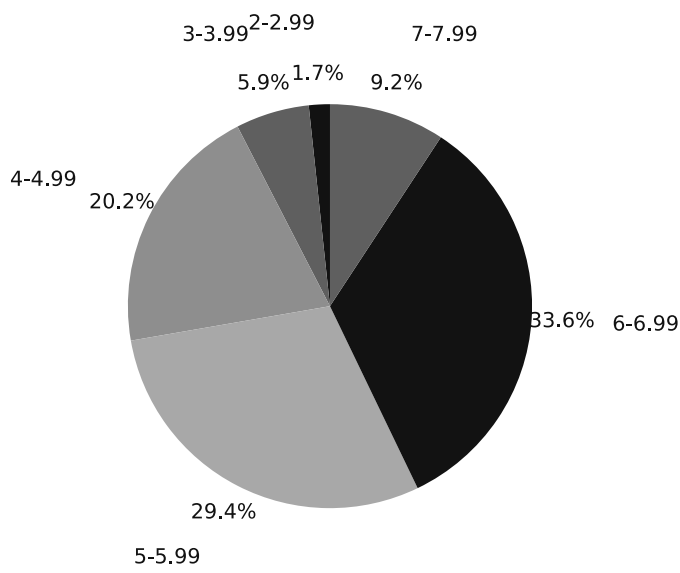


FIGURE 3.7
Pie chart: relative frequency of ladder score ranges.

a limited number of categories, typically fewer than 5-6. In cases with more extensive lists of categories, it is advisable to explore alternative chart types, such as bar charts, for improved clarity. This is because pie charts rely on comparing category sizes, angles, and areas, which can be challenging for the human eye to accurately assess, particularly when numerous categories are involved. To mitigate this issue, it is a recommended practice to accompany pie charts with numeric values for the corresponding categories or slices.

3.1.2 Measures of Central Tendency

Central tendency aims to identify the most representative value of a variable, and this notion of “representative-ness” can be quantified in various ways. In this section, we will delve into three fundamental metrics for central tendency: mode, mean, and median.

Mode

The mode is the value within a variable that occurs with the highest frequency, either in absolute or relative terms. Nominal and ordinal variables are typically analyzed using the mode. To illustrate, in the World Happiness Report dataset, as shown in Table 3.1 and Figures 3.2 and 3.6, the mode for the region variable is Europe and Central Asia.

For continuous variables, identifying the mode can be less practical since these variables often contain unique values with no repetitions. To make the mode meaningful for continuous variables, it is advisable to convert them into categorical or ordinal variables.

The specific modal category is subject to the categorization chosen. For instance, in [Figures 3.3, 3.4, and 3.5](#), different categorizations of the Ladder score (intervals or bin widths) lead to variations in the modal category. Therefore, this approach should be used when valid justifications for the categorization criteria can be provided.

Mean

The mean, often referred to as the “average,” is the simple arithmetic average of a variable’s values. It is calculated by summing all values in the variable and dividing this sum by the total number of values.

Because the mean takes into account every observation in a variable, it can be considered a representative measure of central tendency. It is commonly used for continuous variables and is applied to ordinal variables as well. For example, economists frequently use the mean of GDP per capita (average GDP per person) to assess economic development. Mean values are also employed to gauge product quality, customer satisfaction, and public opinion on various topics.

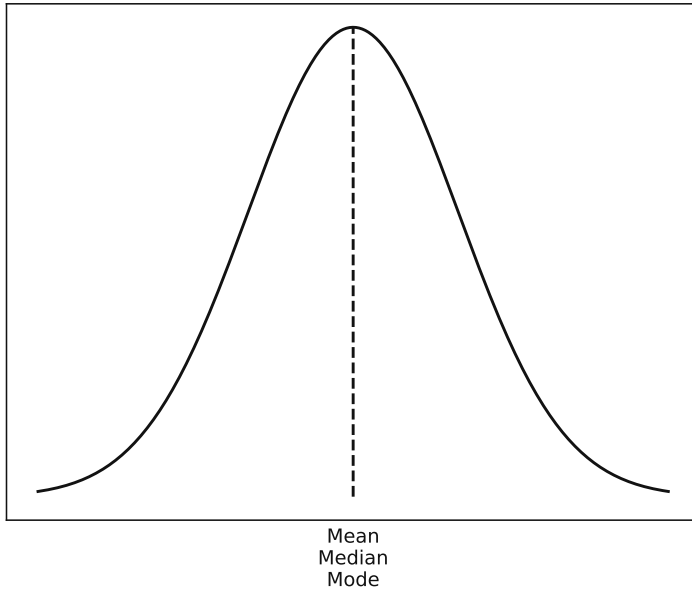
Median

The median is the value that divides a dataset into two equal halves, with one-half of the observations falling below the median and the other half above it. When a variable contains an odd number of observations, the median is the middle value when the variable is sorted in either ascending or descending order. In cases of an even number of observations, the median is the mean of the two neighboring values located in the middle of the variable.

Similar to the mean, the median can be applied to ordinal and continuous variables, provided that the variables are rankable. However, unlike the mean, which considers all values in a variable, the median is determined by only one or two observations situated in the middle of a sorted variable.

3.1.3 Distribution Skewness and the Choice of Central Tendency Metrics

Which of the three central tendency metrics should we choose depends on the variable type and the shape of the variable distribution. For instance, in a symmetrical distribution, the mode, mean, and median are either the same or very close ([Figure 3.8](#)). However, in a skewed distribution, these three metrics will differ. [Figures 3.9 and 3.10](#) illustrate the positions of the mode, mean, and median in symmetrical and skewed distributions.

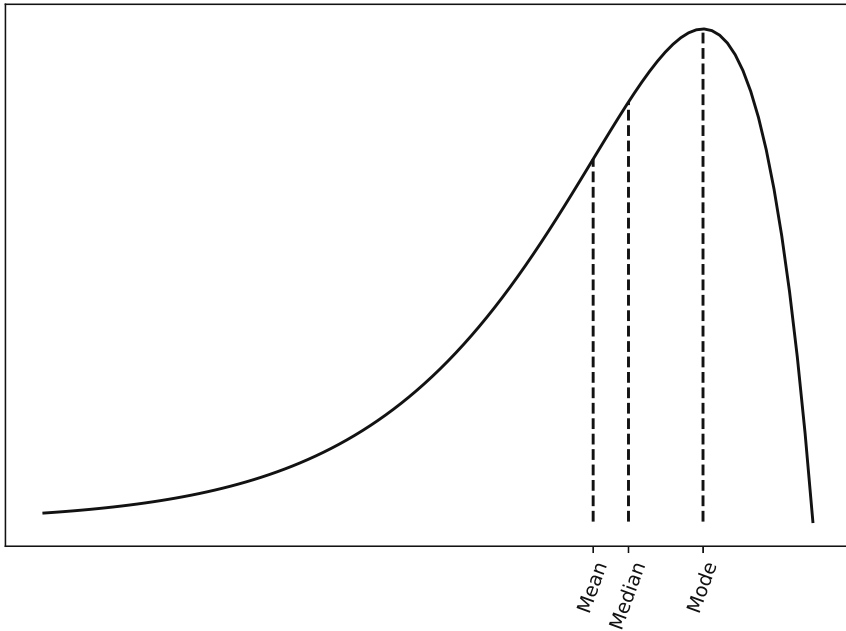
**FIGURE 3.8**

Symmetrical distribution ($N(0,1)$).

In a skewed distribution, the mode, which indicates where the frequency is the highest, is found at the peak. The mean, on the other hand, is pulled (distorted) towards the extreme values in the distribution due to its calculation method. In a left-skewed (negatively skewed) distribution, the mean is shifted to the left, while in a right-skewed (positively skewed) distribution, the mean is shifted to the right. The median of the distribution lies between the mode and the mean.

Comparing these three metrics can offer insights into the shape of the distribution. For example, according to the US Census Bureau, the median US household income in 2021 was \$69,717, while the mean household income was \$97,962. The mode income category was “between \$50,000 and \$74,999.” This result suggests that households with extremely high incomes have a distorting effect on the mean value. Therefore, the Census Bureau and many economists prefer to use median income as a measure of the central tendency of “average” income in the US since it represents the income level that separates half of the population. However, if the goal is to represent everyone equally, then the mean income can be a more suitable metric.

The skewness score of a distribution can range from negative to positive infinity. A skewness value of 0 indicates a symmetrical distribution. The more

**FIGURE 3.9**

Left skewed distribution ($\Gamma(2, 1)$).

extreme the skewness score, whether negative or positive, the more skewed the distribution is.

Multi-modal Distributions

It is important to note that the three central tendency metrics are not always reliable. The examples mentioned earlier typically involve unimodal distributions, which have a single peak. However, in practice, you can encounter distributions with multiple peaks, known as multimodal distributions.

Figure 3.11 illustrates a multimodal distribution with two peaks, making it a bimodal distribution. This distribution could represent the grades of a hypothetical class, where one group of students receives grades around 90 (the right peak in the figure), while another group performs poorly and gets grades around 30 (the left peak). Since the distribution is symmetrical, both the mean and median values are situated in the middle of the distribution, approximately around 60. It is evident that 60 does not accurately represent the class performance.

When dealing with multimodal distributions, trying to summarize the central tendency of the distribution using a single metric can be challenging. In such cases, it is often more appropriate to offer a more nuanced description of

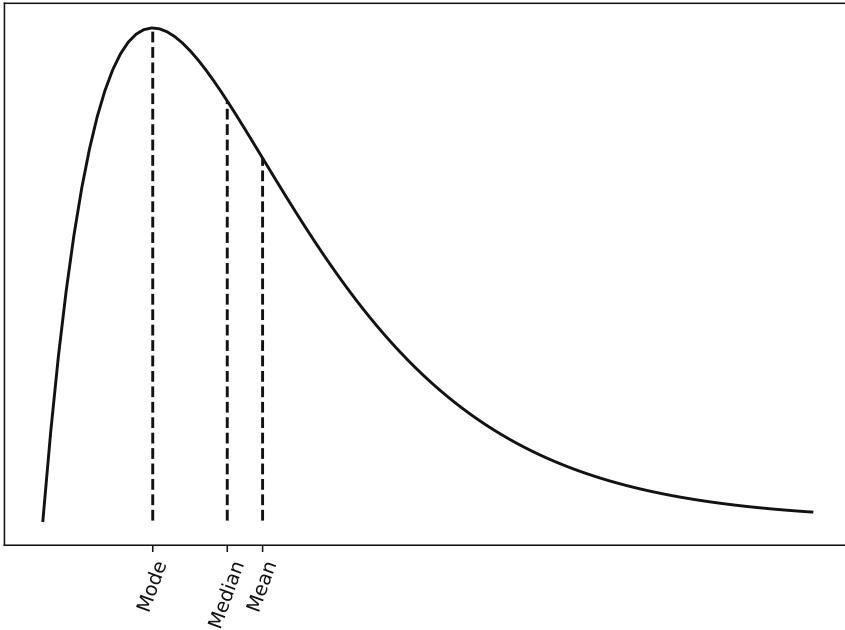


FIGURE 3.10
Right skewed distribution ($\Gamma(2, 1)$).

the distribution. For instance, you can classify the students into two groups (as discussed in [Chapter 6](#) on p. 101) and analyze them separately to gain a better understanding of the data.

3.1.4 Variability

Variability is a measure that quantifies how much the observations or values of a variable spread out. It complements central tendency by providing insight into the dispersion of data in a distribution.

The range is a simple measure of variability, defined as the difference between a variable’s maximum and minimum values. However, it has a limitation: it only considers the two most extreme values and overlooks the variability within the rest of the observations. Therefore, the range alone doesn’t inform us about the distribution’s characteristics, such as whether most observations are concentrated around one value with a single extreme value on the other side or if the observations are evenly spread throughout the range. In essence, the range is an easily calculated but crude measure of variability.

Variance is a more comprehensive metric that takes all observations, the mean, and the number of data points into account. It is calculated by summing

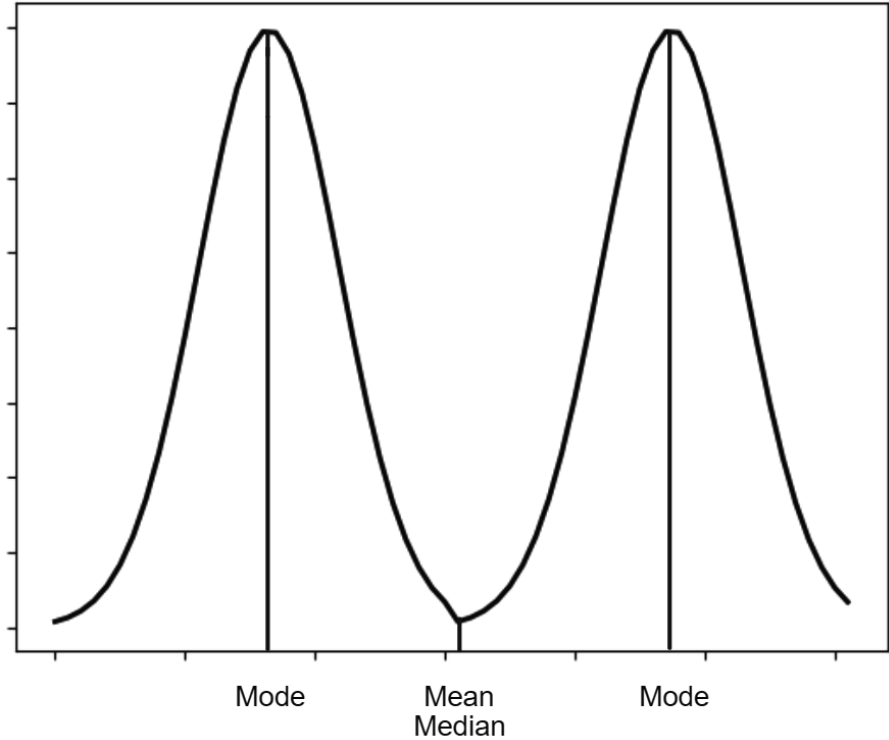


FIGURE 3.11
Multimodal distribution (two superimposed normal distributions).

the squared differences between each data point and the mean, divided by the number of data points. Variance describes the average distance between observations and the variable’s mean. However, because variance involves squared values, its unit differs from the original variable’s unit. For example, if the variable represents distance in meters, its variance would be measured in square meters. This makes the interpretation of variance less straightforward. Additionally, the squaring operation in the calculation makes variance sensitive to outliers.

The standard deviation, on the other hand, is a more commonly used metric for measuring variability. It is the square root of the variance, which means its unit remains the same as that of the original variable. For instance, if the variable measures distance in meters, the standard deviation is also measured in meters, making it easier to interpret. [Figure 3.12](#) illustrates two normally distributed variables with different standard deviations. The upper distribution, represented by dots, has a standard deviation of 1, while the lower distribution, marked with “x” signs, has a standard deviation of 5. As observed, the values in the upper distribution are more tightly clustered compared to

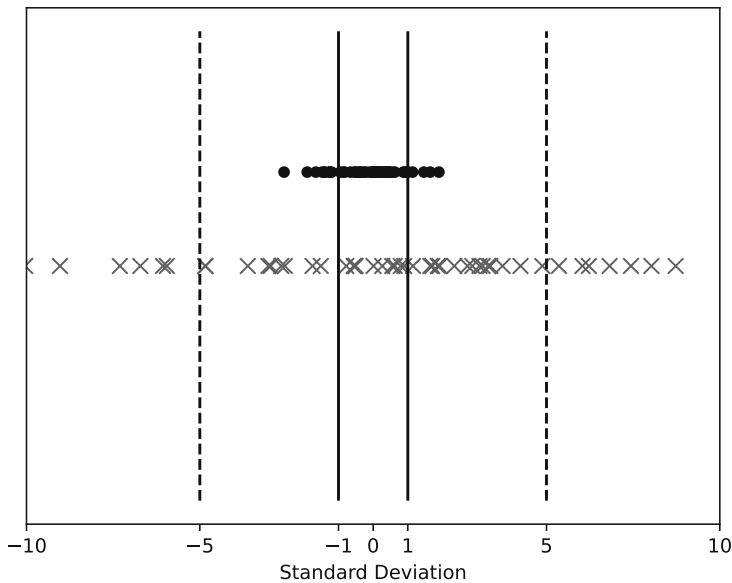


FIGURE 3.12
Standard deviation.

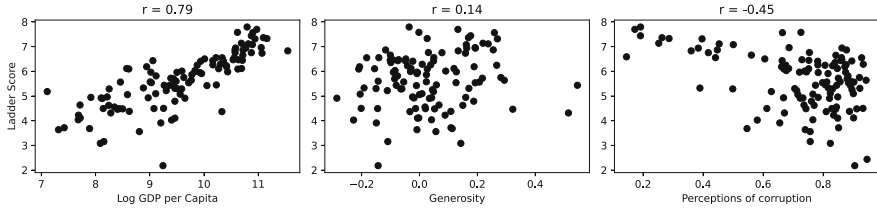
the lower distribution, demonstrating how standard deviation measures the spread or variability of data in a distribution.

Identify Outliers with Standard Deviation

Outliers are data points whose values deviate significantly from the rest of the dataset. But how significant is “significant enough”? Standard deviation, which quantifies the spread of observations around the mean, can help establish a threshold for identifying outliers. While there is no fixed rule for setting this threshold, a common convention is that observations in a large sample following a normal distribution can be labeled as outliers if they deviate by more than three standard deviations from the mean. This threshold provides a rough guide for identifying extreme values.

3.2 Statistics in Exploratory Analysis

In exploratory analysis, we shift our focus from individual variables to examining relationships between them. One crucial metric for this purpose is

**FIGURE 3.13**

Correlations between ladder score and other variables.

correlation, which quantifies the strength of a linear association between two variables. The correlation coefficient, which ranges from -1 to 1, measures this relationship.

A positive correlation (coefficient > 0) suggests that the two variables change in the same direction, while a negative correlation (coefficient < 0) indicates they change in opposite directions. A correlation of 0 signifies independence between the variables. What is considered a “strong” correlation varies by field. In the social sciences, a correlation coefficient of 0.5 or higher can be deemed a strong positive association, whereas -0.5 or lower indicates a strong negative association. However, other disciplines, such as business management and technology, might have different thresholds, depending on their specific needs and contexts. For instance, a correlation coefficient of 0.3 could be considered strong in some medical fields whereas a correlation of 0.95 could be deemed as too low in testing self-driving cars.

Figure 3.13 displays three scatter plots depicting relationships between the Ladder score, logged GDP per capita, generosity, and perception of corruption in the World Happiness Report dataset. A scatter plot visually represents the connection between two variables on a coordinate plane. Each dot on the plot corresponds to one observation, with its coordinates based on the values of the two variables for that observation. Scatter plots are valuable for visualizing variable relationships and assessing correlation.

In the left scatter plot, we observe the relationship between the Ladder score and logged GDP per capita, representing economic development. This relationship exhibits a high correlation of 0.79, signifying a strong positive association. This indicates that countries with higher GDP per capita tend to have higher Ladder scores, reflecting increased well-being.

Moving to the middle scatter plot, we examine the relationship between the Ladder score and generosity. Here, the correlation is only 0.14, indicating a weak association between the two variables. In other words, there is little connection between a country’s Ladder score and its level of generosity.

Finally, the right scatter plot illustrates the relationship between the Ladder score and the perception of corruption. This relationship has a correlation of -0.45, suggesting a moderate negative association. In practical terms,

countries with higher levels of perceived corruption tend to have lower Ladder scores, indicating reduced well-being.

It is crucial to emphasize that **association, as measured by correlation, does not imply causation**. Association simply means that there is a discernible pattern in the variations of two variables, either positively or negatively. It does not provide insights into whether changes in one variable directly “cause” changes in the other variable. The exploration of causality will be a focus of [Chapter 4](#) on p. 52.

In real-life scenarios, it is not uncommon to encounter variables with surprisingly high levels of association that lack genuine causal relationships. For instance, consider the remarkable correlation of 0.997 between the amount of US governmental spending on science, space, and technology and the number of suicides by hanging, strangulation, and suffocation [\[Vig\]](#). Similarly, the correlation of 0.66 between the number of people who drowned by falling into a pool and the number of films featuring Nicolas Cage may seem surprising. However, these high correlation values are spurious because other factors (variables) have a more direct and reasonable influence on the observed outcomes.

It is essential to emphasize that correlation analysis assesses linear relationships between variables. When dealing with non-linear relationships, the correlation coefficient is not a reliable indicator of association. For example, in [Figure 3.14](#), it is clear that the X and Y variables exhibit a non-linear association, yet the correlation coefficient of this relationship is close to 0. This illustrates why it is considered good practice to visually inspect the relationships between variables using scatter plots before conducting, or deciding to conduct, correlation analysis.

Additionally, it is worth noting that when two sets of data have the same correlation coefficient values, it does not necessarily imply that they possess

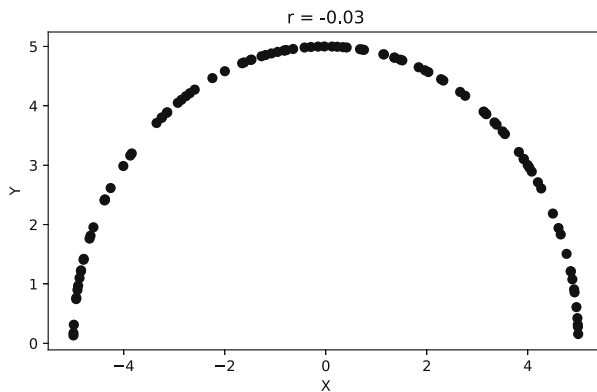


FIGURE 3.14

Correlations of a non-linear relationship.

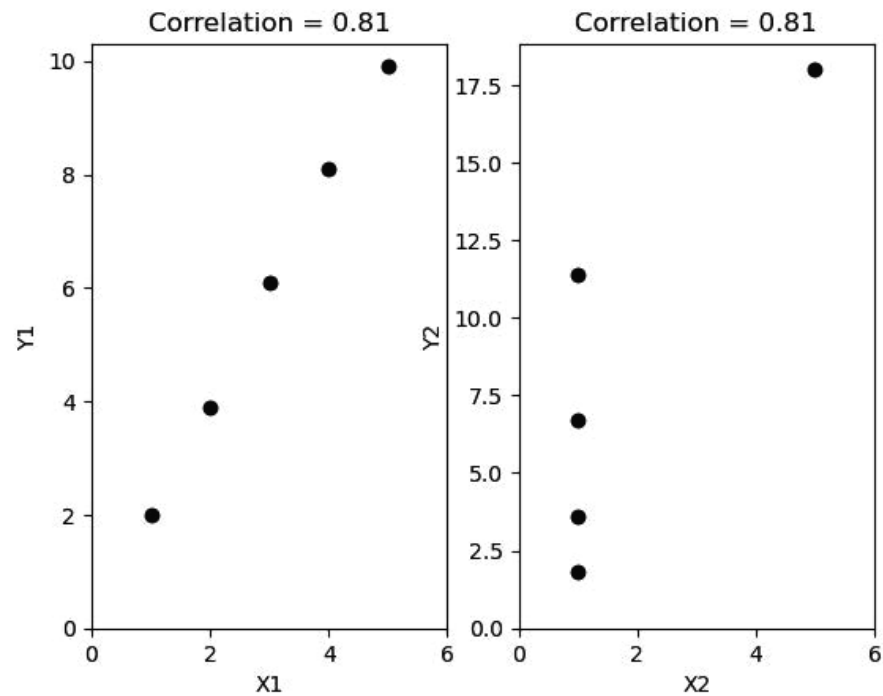


FIGURE 3.15
Different associations with the same correlation coefficients.

the same or similar association patterns. For instance, in [Figure 3.15](#), both sets of data have identical correlation coefficients of 0.81, but it is evident that their association patterns differ substantially. This underlines the importance of not relying solely on correlation coefficients and, instead, examining the scatter plots and the context of the data for a comprehensive understanding of the relationship between variables.

3.3 What’s Next?

Descriptive analysis enables us to grasp the central tendencies of variables, while exploratory analysis reveals their associations. Both of these are crucial steps on the path to causal analysis. In the upcoming chapter, we will delve into the intricate topic of causality, exploring how to construct and scrutinize causal relationships. For those eager to delve into data visualization and the statistical methods covered in this chapter, detailed guidance can be found in

[Chapter 11](#) on p. 217. There, you will also learn how to introduce conditions into your Python program and deal with errors.

3.4 Discussion Questions

1. Identify a phenomenon that you are interested in. How will you measure it as a variable and which metric of central tendency would you use to measure the typical observation of the variable?
2. What type of figure would you use to visualize the frequency distribution of the variable?
3. How would you measure the variability of the variable?
4. Why is it important to consider non-linear relationships when analyzing correlations between variables, and what are the potential limitations of using correlation coefficients in such cases?
5. Why is association not necessarily causation? Can you use an example to support this argument?

Causality and Hypothesis Testing

Causality stands as one of the fundamental concepts in social science. In stark contrast to mere correlation, causality delves into the intricate relationship between two variables. It goes beyond establishing an association, seeking to unveil the dynamic where changes in one variable not only align with changes in another but also instigate those very changes. For a relationship to be deemed causal, it must satisfy several conditions:

1. **Pattern of Variation:** First and foremost, a causal relationship is characterized by a discernible pattern of variation. This pattern can take various forms, be it linear or non-linear.
2. **Temporal Precedence:** Causality insists on temporal precedence — the cause must invariably precede the outcome in time. This chronological order is crucial in establishing the direction of influence.
3. **Absence of Spuriousness:** Beyond temporal alignment, a causal relationship must not be spurious. Spurious connections may appear causal on the surface but are, in reality, driven by a hidden third variable confounding the interpretation. Such relationships can be unmasked through sustained observation and rigorous testing.

This chapter is dedicated to exploring the quantitative methods for identifying and testing causal relationships. In scientific research within the realm of social science, the goal extends beyond uncovering patterns of association. It pivots towards understanding the mechanisms that drive societal changes and the actions of social actors. This chapter will equip you with the tools and techniques necessary to delve into the causal fabric of the complex social world.

This chapter is dedicated to exploring the quantitative methods for identifying and testing causal relationships. In scientific research within the realm of social science, the goal extends beyond uncovering patterns of association. It pivots towards understanding the mechanisms that drive societal changes and the actions of social actors. This chapter will equip you with the tools and techniques necessary to delve into the causal fabric of the complex social world.

4.1 Randomized Controlled Experiments

A foundational principle in identifying causal relationships is to consider counterfactuals. To illustrate, during my childhood, I often caught the flu, prompting my mother to give me various medicines, assuring me that they were effective in curing the illness. However, I later discovered that many flu cases naturally resolve themselves within a week without any medication [God].

This raises an important question: To what extent can I attribute the flu's recovery to the medicine I took? Could it be that the medication had no real impact, and the flu simply ran its course? The challenge lies in the unobservable counterfactual. We cannot directly perceive what would have occurred if I had not taken the medicine.

Fortunately, in the field of pharmaceutical research, causal effects are a paramount concern. When determining whether a medication can effectively treat a particular disease, a clinical trial design that involves administering the medicine to a group of patients and observing their outcomes is not enough. This approach falls short because it fails to address the essential question: What would have happened to these patients had they not received the medication?

One prominent approach to establish causal relationships is through a **randomized controlled experiment**. Here is how it works:

1. We begin by randomly selecting patients who share a common ailment.
2. Then, we randomly divide these patients into two distinct groups: a treatment group and a control group.

In the treatment group, patients are administered the medication, whereas the control group either receives no medication or is given a placebo. This approach ensures that patients' outcomes are not influenced by psychological factors. To gauge the causal impact of the medication, we analyze and compare the average outcomes between these two groups. If patients in the treatment group exhibit better outcomes than those in the control group, it provides evidence for the medication's effectiveness (the degree of effectiveness, however, is another consideration, to be discussed further below).

It is important to note that even in such experiments, we still cannot directly observe the elusive counterfactuals—namely, what would have happened if patients in the treatment group had refrained from taking the medicine or if those in the control group had taken it. Nevertheless, the process of randomly selecting patients and assigning them to their respective groups ensures that, despite their inherent differences, the two groups are comparable in terms of their characteristics before receiving treatment. In simpler terms, the sole distinction between the treatment and control groups is the presence or absence

of the treatment. Consequently, any differences observed in the outcomes between these groups can be attributed exclusively to the treatment itself.

4.1.1 Limitations of Randomized Controlled Experiments

While randomized controlled experiments are a powerful tool for establishing causality, they come with certain limitations. These experiments necessitate a high degree of control and intervention by researchers. On the positive side, this level of control makes them effective for pinpointing causal relationships. However, the same high level of control that makes randomized controlled experiments effective also restricts the generalizability of their findings. The carefully selected samples may not accurately represent the broader population.

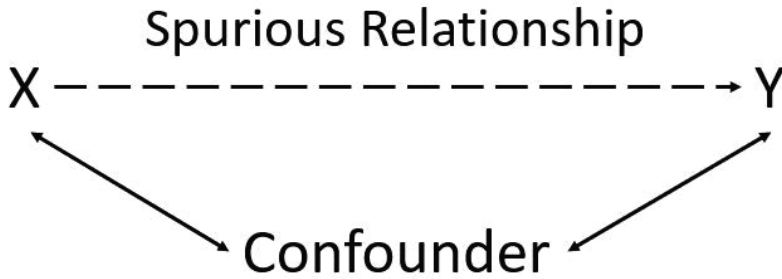
In essence, the trade-off is between precision in establishing causality and the ability to extend the findings to a larger context. This means that while the results of such experiments may be conclusive within their specific settings, applying those findings to a more extensive or diverse population may not be straightforward.

4.2 Observational Methods

In many social science research scenarios, conducting randomized controlled experiments is neither feasible nor ethical due to logistical and practical constraints. Large-scale studies involving human subjects can be particularly challenging and costly. This section explores observational methods that social scientists commonly employ when experimentation is not a viable option.

Researchers in fields such as political science often face hurdles in conducting randomized experiments. For example, imagine a study aiming to investigate the impact of political campaign ads on voter turnout. In this case, researchers would need to create TV ads, purchase airtime, and selectively expose viewers to these ads while keeping others unexposed. The complexity and cost of this task can be overwhelming. Moreover, ethical considerations can render random assignment impractical. For instance, studying the impact of authoritarian political systems on various social and economic outcomes through random assignment involves assigning people to live under authoritarian political environment for a long period of time. This is obviously ethically problematic, not to mention that few people would participate in such experiment.

These challenges lead researchers to rely on observational methods, which involve gathering data from naturally occurring situations rather than manipulating variables through experiments. While these methods do not offer the same level of control as experiments, they remain valuable tools in social

**FIGURE 4.1**

A confounder in a spurious relationship.

science research. A study of political communication and political campaign aims to analyze the impact of political advertisements on the target audience [Lla18]. The study compared areas that were inadvertently exposed to an intense influx political advertising to areas that did not experience such exposure.

Observational studies offer the advantage of increased generalizability as their subjects are typically more representative of the broader population. However, it is crucial to ensure comparability between the treatment and control groups, with the goal of having treatment as the only differing factor. Researchers must also remain vigilant regarding the presence of **confounders**, which are variables existing before treatment and associated with both the treatment and outcome variables (Figure 4.1).

For instance, consider a study on political campaign advertisements where factors like age, employment status, and economic standing exist prior to the study. These factors can influence an individual's likelihood of viewing political campaign ads on TV and their likelihood of voting. In this context, they could potentially affect the relationship between TV ads and voter turnout, complicating the analysis of a causal link.

In certain cases, confounding bias may stem from selection bias, which arises due to researchers lacking control over their study subjects' choices regarding whether to participate in the treatment group. Such bias can emerge when those opting for treatment significantly differ from those declining participation or receiving treatment. For instance, research into the factors contributing to the success of startup companies may encounter the risk of solely including prosperous startups while excluding failed ones (a scenario known as survival bias). Consequently, the factors identified as the driving forces behind successful startups may not have a causal influence on their achievements, masking the true underlying causes. Likewise, in studies reliant on data

gathered through interviews and surveys, observations are limited to individuals who willingly participate. The respondents or interviewees might differ from those who choose not to participate.

4.2.1 Solutions to Confounders

While confounding effects cannot be entirely eradicated from observational studies, researchers can employ subclassification as a strategy to mitigate their impact. This approach involves comparing the treatment and control groups within subsets of the sample, defined by shared values in the confounding variable.

Consider the study of political campaign ads mentioned earlier, where age and economic status are pre-existing factors that may confound the analysis. While these factors cannot be eliminated, researchers can compare individuals who were exposed to campaign ads and those who were not within the same age group or economic status category. By doing so, the researchers can effectively filter out the confounding effects and obtain a clearer view of the relationship between independent and dependent variables.

4.3 Hypothesis and Hypothesis Testing

Once a relationship or pattern among variables has been identified, the question arises: how confident can we be in our findings? Is it possible that, despite our rigorous research design, the observed patterns are merely a result of chance?

To address these concerns, we turn to hypothesis testing. A hypothesis is a statement about the world that we aim to validate through empirical evidence. In scientific research, hypotheses must be testable, and hypothesis testing is a critical component of causal analysis. However, the process of hypothesis testing is not always straightforward. In scientific exploration, we rarely possess all the necessary information to directly prove or disprove our hypotheses. This is due to the inherent limitations of human knowledge, and our understanding of the world is continually evolving with new discoveries.

In hypothesis testing, we do not set out to confirm our hypothesis directly; instead, we seek to test the absence of a relationship or pattern, which is known as the **null hypothesis** (denoted as H_0). The null hypothesis suggests the absence of an effect, relationship, or difference. The original hypothesis that we believe to be true is referred to as the **alternative hypothesis** (denoted as H_a). The primary reason for focusing on the null hypothesis is that, while we may not be able to definitively prove that one variable causes changes in another, it is relatively easier to examine whether there is no relationship between them.

4.3.1 Step 1: Establish the Null Hypothesis

The process of hypothesis testing, as depicted in Figure 4.2, begins with the first step: establishing the null hypothesis based on the research question. The structure of a research question influences the formulation of hypotheses in two ways: the number of groups under consideration and the direction of the inquiry. Table 4.1 provides examples of research questions and their corresponding null and alternative hypotheses.

When it comes to the number of groups involved, a research question focusing on a single population or group directly provides the parameter and its value under the null hypothesis, which is also referred to as the null parameter. For example, if the research question pertains to whether a class has a mean grade of 80, then the null parameter would be the mean grade, with a value of 80. However, when a research question involves comparing more than one group, such as evaluating whether two classes or a single class before and after a specific treatment are distinct, the null parameter would represent the difference between these groups, with a parameter value of 0.

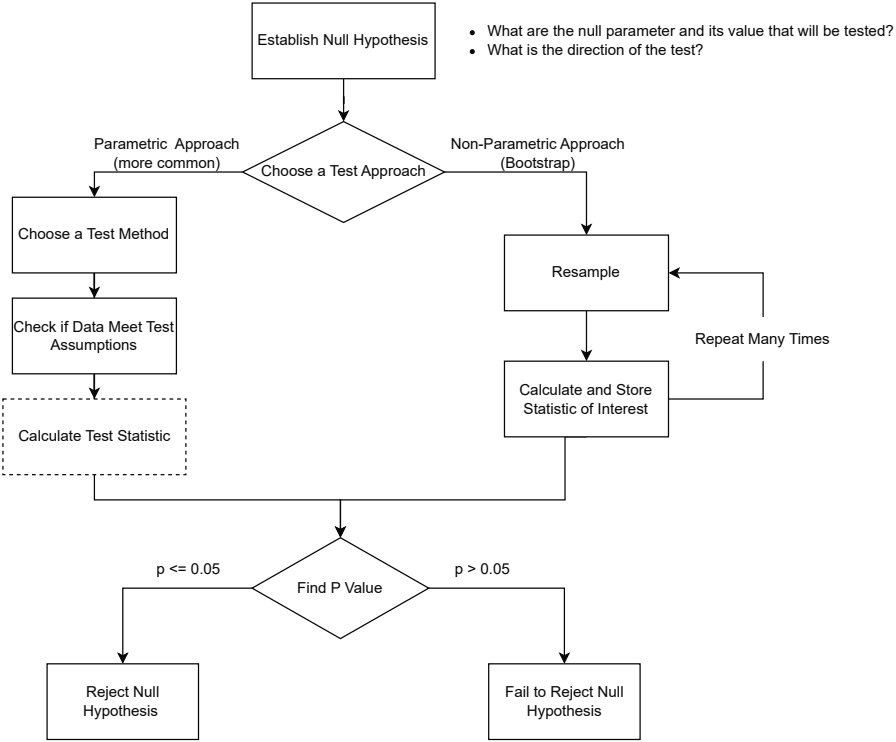


FIGURE 4.2
Hypothesis testing process.

TABLE 4.1
Writing a hypothesis based on research question

Research Questions About One Group		
Non-directional (Two-tailed)	Is the mean grade of a class different from 80?	H_0 : The mean grade of a class is not different from 80. H_a : The mean grade of a class is different from 80.
Directional (Right-tailed)	Is the mean grade of a class higher than 80?	H_0 : The mean grade of a class is not higher than 80. H_a : The mean grade of a class is higher than 80.
Directional (Left-tailed)	Is the mean grade of a class lower than 80?	H_0 : The mean grade of a class is not lower than 80. H_a : The mean grade of a class is lower than 80.
Research Questions About More Than One Group		
Non-directional (Two-tailed)	Is there any difference between two classes in mean grades?	H_0 : There is no difference between the two classes in mean grades. H_a : There is a difference between the two classes in mean grades.
Directional (Right-tailed)	Is the mean grade of class 1 higher than that of class 2?	H_0 : The mean grade of class 1 is not higher than that of class 2. H_a : The mean grade of class 1 is higher than that of class 2.
Directional (Left-tailed)	Is the mean grade of class 1 lower than that of class 2?	H_0 : The mean grade of class 1 is not lower than that of class 2. H_a : The mean grade of class 1 is lower than that of class 2.

The nature of a research question can also influence the formulation of hypotheses by determining whether the question is non-directional or directional. Non-directional research questions refrain from specifying the direction (positive or negative) of the relationship between variables prior to conducting a test, while directional research questions explicitly state the direction.

Non-directional research questions, which are more common in academic research, center around determining whether values of interest are “different” from one another. The null hypotheses associated with non-directional research questions include phrases such as “not different,” “no difference,” or “equal to.” Consequently, the alternative hypotheses incorporate phrases such as “is different,” “has difference,” or “not equal to.”

For instance, in a test to determine if the mean grade of a class differs from 80, the null hypothesis posits that the mean grade is “not different” from (or

“equal to”) 80, with the alternative hypothesis asserting that “the mean grade of the class is different from (or not equal to) 80.” Similarly, for a test investigating whether patients in the treatment group exhibit different outcomes than those in the control group, the null hypothesis reads as “the treatment group and the control group have no difference in outcomes,” implying that the mean difference in outcomes between the two groups is 0. In contrast, the alternative hypothesis states that “the treatment group and the control group are different in outcomes.”

Non-directional research questions are designed this way because we often lack prior knowledge about whether the class average grade is above or below 80, or whether the treatment would improve or worsen patients’ symptoms. Such hypothesis testing is also known as a “two-tailed test” because it encompasses and tests two different possibilities.

Alternatively, directional research questions can be used to test either the positive or negative aspect of the possibilities. This type of hypothesis testing is referred to as a “one-tailed test,” which includes right-tailed and left-tailed tests. A right-tailed test assesses whether the value of the parameter of interest is greater than another value (as a convention, higher values are on the right side of lower values on a horizontal scale).

Here is an example of a right-tailed test: To determine whether the mean grade of a class is higher than 80, rather than just different from 80, the null hypothesis would state that the mean grade is “not higher” than 80, which could be 80 or below. The alternative hypothesis then asserts that “the mean grade of the class is higher than 80.” Similarly, instead of evaluating whether two groups of patients differ in outcomes, researchers may inquire whether the treatment group exhibits superior outcomes compared to the control group. In this case, the null hypothesis asserts “the treatment group does not have better outcomes than the control group,” implying that the mean difference in outcomes between the two groups is 0 or less. The alternative hypothesis, on the other hand, posits that “the treatment group has better outcomes than those who do not.”

In contrast, a left-tailed test addresses whether the mean grade of the class is “lower” than 80 or whether the treatment group of patients exhibits worse outcomes than the control group. The null hypotheses for these two research questions are “the mean grade of the class is not lower than 80” and “the treatment group does not have worse outcomes than the control group,” respectively. Meanwhile, the alternative hypotheses are “the mean grade of the class is lower than 80” and “the treatment group has worse outcomes than the control group.”

4.3.2 Step 2: Choosing a Testing Approach

The next step in the hypothesis testing process is to choose between the parametric approach and the non-parametric approach. The selection depends on our understanding of the probability distribution governing the population.

The parametric approach is preferred when we believe or know that a specific probability distribution, with known parameters, accurately describes the population. This approach relies on the parameters that define these probability distributions.

For instance, the normal distribution is one of the most widely used probability distributions, known for its bell-shaped, symmetric characteristics. A normal distribution is characterized by two parameters: its mean and standard deviation. The distribution is centered around the mean value, while the standard deviation dictates the spread of the distribution around the mean (a larger standard deviation results in a more dispersed distribution). A standard normal distribution has a mean of 0 and a standard deviation of 1. Other commonly used probability distributions include the t-distribution (also known as Student's t-distribution), F-distribution, and Chi-squared distribution. The mathematical details of these distributions are beyond the scope of this book, and we will focus on how to select distributions and corresponding test methods for analysis in the next section.

In contrast to the parametric approach, the non-parametric approach, also known as the distribution-free or model-free approach, does not assume a specific probability distribution for the population or rely on distribution parameters. Instead, it estimates the population by generating new random samples from the observed data. The non-parametric approach is chosen when we have limited knowledge of the population probability distribution or when the sample data fail to meet the assumptions of parametric tests.

The choice between these approaches depends on several factors. One criterion to consider is whether we can make any assumptions about the population distribution. Additionally, both approaches come with their own advantages and disadvantages. The non-parametric approach offers greater flexibility since it is not constrained by assumptions about the probability distribution, making it more reliable when data do not meet test requirements. However, when data do meet these requirements, the parametric approach can yield more accurate results. Furthermore, the non-parametric approach is computationally more intensive than the parametric approach because it avoids simplifying assumptions about data distribution. This disadvantage becomes more apparent with large datasets and complex models.

In the field of social sciences, parametric approaches are commonly used. Therefore, in the following section, we will explore some widely employed parametric test methods, their appropriate use cases, and the data requirements for these tests.

4.3.3 Step 3: Selecting a Test Method, Calculate the Test Statistic, and Find the p-value

The choice of a test method is guided by the probability distribution assumed for the population. This choice subsequently dictates the test statistic used to evaluate the null hypothesis. A test statistic is a numerical value derived

from sample data, assisting in the determination of whether there is sufficient evidence to support or reject a hypothesis about a population parameter. Although the specific calculation of test statistics varies between methods, the general procedure involves (1) finding the difference between a sample statistic (e.g., the mean grade of a class or the mean difference in outcomes between treated and untreated patients) and the null parameter and (2) dividing it by the standard error of the sample statistic. The standard error measures the variability of a sample statistic, like the sample mean. It quantifies how much the sample statistic is expected to deviate from the true population parameter. Unlike the standard deviation, which measures data variability within a sample, the standard error estimates variability among samples, even if we do not have all the samples from the population. A larger standard error indicates more variability among samples.

Modern statistical software libraries such as ``scipy`` and ``statsmodels`` in Python handle the computation of test statistics. However, it is essential to understand that the test statistic's value signifies the number of standard errors between the sample statistic and the null parameter.

As the test statistic value increases, the discrepancy between the sample statistic and the null parameter also intensifies. A very large test statistic indicates that the observed data significantly deviates from what would be expected under the null hypothesis. In such cases, where the null hypothesis contradicts real-world observations, we must **reject the null hypothesis**. Conversely, if the test statistic is very small, it means the sample data does not significantly differ from what we would expect under the null hypothesis. In this scenario, where the sample aligns with the null hypothesis, we **fail to reject the null hypothesis**.

It is essential to note that we can only state that we “fail to reject” a null hypothesis but not “accept” it because the result merely suggests we lack sufficient evidence to claim that our observations significantly differ from the null hypothesis. In line with the philosophy of science, we should remain open to new information and findings. When multiple studies reach the same conclusion with different samples, the conclusion gains greater reliability.

To reject a null hypothesis, we require the difference between the test statistic and the null parameter to be statistically significant (specifically, a large test statistic). The determination of whether a difference is statistically significant is based on the p-value.

The p-value

The **p-value** represents the probability of obtaining a test statistic as extreme or more extreme than the one observed, assuming the null hypothesis is true. When this probability is equal to or lower than a predefined threshold, it indicates that the difference between the observed data and the null parameter is statistically significant, leading to the rejection of the null hypothesis.

Conversely, if the probability is higher than the threshold, the difference is not considered statistically significant, and we fail to reject the null hypothesis.

Conventionally, the p-value threshold for statistical significance is set at 0.05 or 5%. This means that most researchers consider a 5% probability or lower to be extremely low. However, it is important to note that the choice of this threshold is somewhat arbitrary. Arguments can be made that 6% or 9% might also be considered extremely low probabilities. Just as individuals have varying levels of risk tolerance, different academic disciplines also differ in their choices of p-value thresholds. For example, researchers in social science fields might reject a null hypothesis when the p-value is less than 0.1 (10%), while disciplines like biology or medical science may require p-values of 0.01 (1%) or even 0.001 (0.1%) to ensure that the observations are highly improbable under the null hypothesis and that the analysis conclusion is robust.

The generation of the p-value is closely related to the test statistic and the assumed probability distribution. In the case of normal distributions, statisticians have established the “68–95–99.7 rule.” This rule states that if a variable follows a normal distribution, approximately 68%, 95%, and 99.7% of observations fall within one, two, and three standard deviations of the mean, respectively (Figure 4.3) [Ker14]. In the figure, σ represents the standard deviation, and μ stands for the mean.

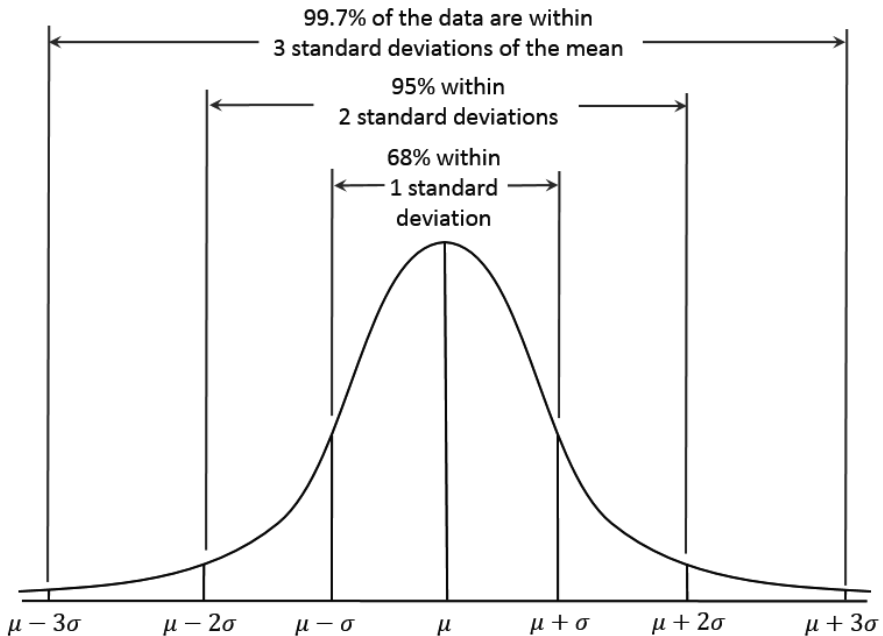


FIGURE 4.3

Normal distribution.

This rule is particularly valuable in hypothesis testing methods that rely on the normal distribution, such as the Z-test (to be discussed in the next section). For instance, consider a two-tailed test (non-directional null hypothesis) using the normal distribution, and the test statistic value is 2. According to the 68–95–99.7 rule, values within 2 standard deviations around the mean account for 95% of all values (technically, it is 1.96 standard deviations, but it is often rounded up to 2 for simplicity). This means that values to the left of $\mu - 2\sigma$ and those to the right of $\mu + 2\sigma$ account for 5% of all values (95% - 5%). In other words, these extreme values on both sides of the tails have a 5% chance of occurring (2.5% on each side) if the null hypothesis is true.

Now, if our test statistic is 2, it implies that it has only a 5% chance of occurring if the null hypothesis is true. Under the conventional p-value threshold of 0.05, the null hypothesis would be rejected.

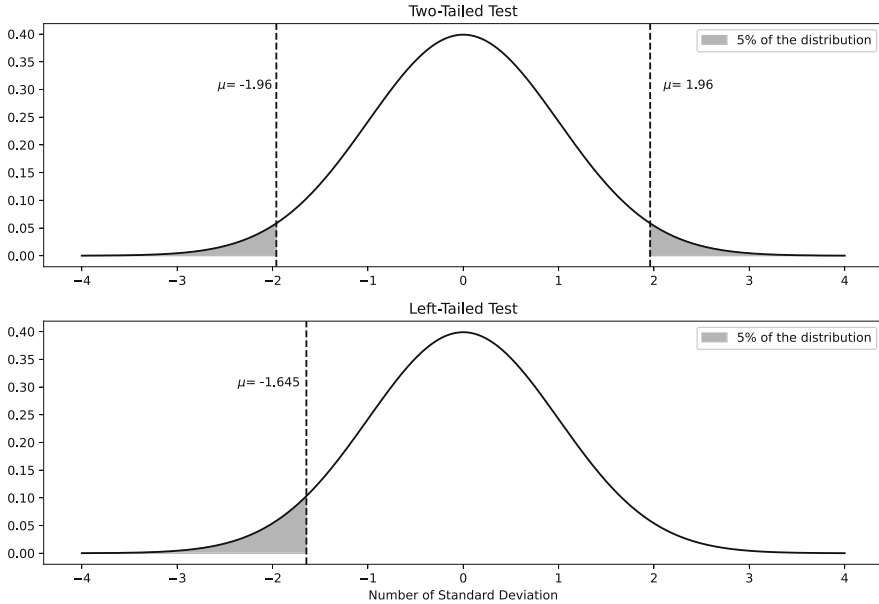
Note that the direction of the null hypothesis can affect how we interpret the 5% threshold. To illustrate, let's consider a left-tailed test. In this case, we focus on the left half of the normal distribution. Continuing to use the 5% p-value threshold, it is important to stress that, unlike in a two-tailed test where the 5% extreme values are distributed equally between the left and right tails, in a left-tailed test, the 5% extreme values only come from the left tail. Based on the probability distribution of the normal distribution, approximately 5% of all values fall more than 1.645 standard deviations to the left of the mean, which corresponds to a test statistic value of -1.645.

Figure 4.4 provides a visual representation of the test statistic thresholds for a two-tailed test and a left-tailed test in a normal distribution. As shown, to achieve statistical significance in a two-tailed test using a normal distribution, the test statistic must be either greater than 1.96 or smaller than -1.96. In a left-tailed test, a test statistic of -1.645 is sufficient to achieve statistical significance (1.645 for a right-tailed test). This indicates that a two-tailed test requires more extreme differences to attain statistical significance and is consequently more stringent than a one-tailed test. This is one of the reasons why the two-tailed test is more commonly used.

4.3.4 Some Common Test Methods

In this section, we will explore some common test methods tailored to different data and analysis scenarios. To illustrate these tests, we will use data from the World Happiness Report for the year 2021. We chose to focus on 2021 data exclusively due to the inherent interdependence of a country's data with its historical records. This interdependence creates a link through time (it is also called time-series data), making the sample data non-independent, which may not be suitable for some of the tests discussed. Time-series analysis is a more advanced topic in data analysis and is beyond the scope of this book. For now, by concentrating on one year of data, we ensure that the tests we discuss are applicable.

In each example, we will not delve into the mathematical procedures for calculating the test statistic and p-values, as these are typically handled by statistical software. Instead, we will emphasize the interpretation of results.

**FIGURE 4.4**

Test statistic for the 5% threshold in two-tailed test vs. in left-tailed test.

Z-test: Compare Sample Mean and Population Mean

The Z-test compares a sample mean with the population mean and tests whether the difference between them is statistically significant. Its null hypothesis states that there is no difference between the sample mean and the population mean.

To apply the Z-test, several requirements must be met:

1. We need to know the population mean and standard deviation.
2. The sample data should be normally distributed. A normality test, such as the Shapiro–Wilk test, is often conducted before applying the Z-test (an example will be provided shortly).
3. The sample size should be sufficiently large (typically more than 30 observations). Fourth, observations must be independent from each other.
4. The data for the Z-test must be continuous. Categorical data require different tests.

The Z-test relies on the normal distribution and generates a test statistic called the Z-score, which measures the number of standard deviations between

the sample mean and the population mean. With the Z-score and the normal distribution, the software can calculate the corresponding p-value.

In the context of the 2021 World Happiness Report data, we can use the Z-test to analyze the difference in happiness levels between Sub-Saharan Africa (sample) and the world (population). The data indicate that the mean Ladder score for the world is 5.63, and the mean score for Sub-Saharan African countries is 4.43. Is Sub-Saharan Africa as happy as the world on average? In other words, is the observed difference in mean Ladder scores statistically significant?

To answer this question, the null hypothesis states that there is no difference in the Ladder score between Sub-Saharan Africa and the world (non-directional null hypothesis). The Z-test is suitable because we have information about the population. However, before applying the Z-test, we must confirm that our data is normally distributed. The Shapiro–Wilk test can be used to test for normality. The null hypothesis of the Shapiro–Wilk test states that the sample data is normally distributed. If the p-value of the Shapiro–Wilk test is greater than 0.05, we fail to reject the null hypothesis, indicating that the data is likely normally distributed. After conducting the Shapiro–Wilk test on the Sub-Saharan African countries, the p-value is 0.82, meaning there is an 82% chance to obtain the observed result if the null hypothesis (values are normally distributed) is true. The Python code used to perform the Shapiro–Wilk test can be found in [Section 12.2.1](#) on p. 239.

With the normality assumption met, we can perform the Z-test (code used for this example is discussed in [Section 12.2.2](#) on p. 240). The test returns a Z-score of 4.65 and a p-value of 0.0. According to the “68–95–99.7” rule, values that are more than 3 standard deviations away from the mean have less than 0.3% chance of occurring. Based on the Z-score of 4.66, the observed data is located 4.66 standard deviations away from the mean and to the left, resulting in a p-value of 0. Therefore, we can reject the null hypothesis and conclude that the difference between the mean Ladder scores of Sub-Saharan Africa and the world is statistically significant and likely not due to chance, even though the values 4.43 and 5.63 may not seem significantly different at first glance.

T-Test: Compare Two Sample Means

The t-test is a tool used to compare means, specifically to compare two sample means. Its null hypothesis states that there is no difference between the two sample means. The t-test has similar data requirements to the Z-test. The data must be continuous, approximately follow a normal distribution, and be randomly drawn from the population.

The t-test relies on the t-distribution, which is another symmetric, bell-shaped distribution but has heavier tails than the normal distribution. Unlike the normal distribution, which is characterized by the mean and standard deviation of data, the t-distribution is characterized by its degrees of freedom.

Degrees of freedom represent the number of independent pieces of information available in the data. As the degrees of freedom increase, the shape of the t-distribution approaches that of a normal distribution. The use of the t-distribution allows the t-test to work with small samples. The test statistic of the t-test is the t-statistic. We can use the t-statistic and the degrees of freedom from the data to estimate the p-value of the t-test.

There are two types of t-tests, depending on whether the two samples belong to the same group of subjects or not. The first type is called an independent samples t-test or a two-sample t-test. As the name suggests, it is applied to two independent samples. For example, in the World Happiness Report, “East Asia and Pacific” and “Middle East and North Africa” are two independent regions (samples). We can compare the mean Ladder scores of the countries in these regions. After conducting descriptive analysis, it is found that the mean Ladder score for East Asia and Pacific countries is 5.78, and that of the Middle East and North Africa is 5.18. Can we claim that the people in Asia and people in the Middle East and North Africa are different in levels of happiness? The null hypothesis is that the two regions are not different.

As we did before with the Z-test, we need to check if the data meet the normality assumption first with the Shapiro–Wilk test. The test on the two samples shows that neither sample violates the normality assumption, which means we can move forward with the t-test. The t-test returns a p-value of 0.18. Since the p-value is greater than 0.05, we fail to reject the null hypothesis. In other words, even though East Asia and the Pacific has a higher Ladder score than the Middle East and North Africa, the difference between the two regions is not large enough to be considered statistically significant. Thus, we do not have enough evidence to claim that the two regions have a meaningful difference in the Ladder score. The observed difference between them could be a result of chance.

The second type of t-test pertains to comparing two samples from the same group of subjects collected at different time points, known as a paired t-test or dependent samples t-test. This form of t-test is particularly valuable for assessing changes within a single sample over time. To illustrate, within the World Happiness Report dataset, we can analyze the mean Ladder scores of East Asian countries in 2011 and 2021, yielding values of 5.84 and 5.88, respectively. With these observed values, we can investigate the alterations in happiness levels within Asia during the decade.

Since both samples originate from the same group of subjects (East Asian countries), we can employ the paired t-test to evaluate the null hypothesis, which posits that there is no disparity in Ladder scores over the ten-year period. In a manner analogous to our previous use of the Z-test, we must first confirm that both samples satisfy the Shapiro–Wilk test for normality. Furthermore, the paired t-test necessitates that both samples have an identical sample size, as it is designed to compare the same subjects before and after a specific time point. Instances with missing values must be addressed through deletion or imputation prior to conducting the test.

In our dataset, Myanmar has a missing value for 2011, and Malaysia has one for 2021. For the purpose of this analysis, these countries are removed from data to ensure uniform sample size. In [Section 12.2.3](#) on p. 241, we will adopt a more advanced technique to impute the missing data with interpolation.

Upon subjecting the data to a paired t-test for the mean Ladder scores of East Asian countries in 2011 and 2021, we obtain a p-value of 0.79. This outcome indicates that we fail to reject the null hypothesis. Consequently, despite a minor decrease in the Ladder score for Asian countries from 2011 to 2021, we cannot assert that this difference is statistically significant. Therefore, we do not possess sufficient evidence to conclude that the happiness levels of Asian countries underwent a meaningful change over the ten-year span.

ANOVA: Compare Means Among Multiple Groups

ANOVA (Analysis of Variance) is specifically designed to compare means among three or more groups. When you have only two groups to compare, the t-test is the more suitable choice. However, if you have more than two groups and you want to determine whether there are any significant differences in means across these groups, ANOVA is the appropriate tool. In fact, the t-test is considered a special case of ANOVA.

In the context of ANOVA, the null hypothesis asserts that there are no significant differences in means among the categories. To conduct an ANOVA test, certain requirements must be met.

1. The dependent variable should be continuous, while the independent variable(s) must be categorical.
2. Observations must be independent and follow a normal distribution.
3. the categories being compared should have equal variance.

The ANOVA test yields an F-statistic and estimates the p-value using the F-distribution, a positively skewed distribution determined by two degrees of freedom.

One notable advantage of ANOVA is that it can accommodate multiple independent variables. When a single independent variable is used to analyze differences between groups, it is referred to as one-way ANOVA. If two independent variables are used, it is known as two-way ANOVA.

In the one-way ANOVA test, the dependent variable is continuous, and the independent variable is categorical. This test is employed to compare the means of the categories with respect to the overall mean. The null hypothesis is rejected if any group means significantly differ from the overall mean. For example, you can compare the mean Ladder scores among three groups: East Asia and Pacific, Europe and Central Asia, and North America. The test result yields a p-value of 0.039. Since the p-value is less than 0.05, we reject the null hypothesis. This means that the mean Ladder scores of the three regions are different. However, it is important to note that a statistically significant ANOVA test result only indicates the existence of a difference

among the groups. It does not provide information about which specific group is statistically different from the others.

When we believe that the dependent variable is influenced by two categorical independent variables, we can employ the two-way ANOVA test. For instance, we can investigate whether a country's mean Ladder score is influenced by both its region and income group simultaneously. In this test, region and income group are considered independent variables, while the Ladder score is the dependent variable.

The results of the test (Python code available in [Section 12.2.4](#) on p. 242) indicate that both independent variables have p-values of 0. With two independent variables, the interpretation of the results is that each variable has a statistically significant effect on the dependent variable while keeping the other variable constant. In other words, for countries within the same region, their mean Ladder scores exhibit statistically significant differences across income levels. Similarly, among countries within the same income group, their mean Ladder scores display statistically significant differences across different geographic locations.

Chi-squared Test of Independence: Compare Frequencies of Categories

The chi-squareds (pronounced as “ky” in “sky”) test of independence is a widely used method to compare the observed and expected frequencies of categorical data. It assesses whether there is any association between two categorical variables. This test assumes that the population follows the chi-squared distribution, which is a positively skewed probability distribution. To apply the chi-squared test, several conditions must be met:

1. The observations in the sample are independent of each other,
2. Each category has a sufficiently large sample size, typically at least five.
3. The data being analyzed must be categorical.

We can use the World Happiness dataset to investigate if a country's income level is associated with its level of perceived corruption. In the dataset, the income group variable classifies countries into one of four categories: low income, lower-middle income, upper-middle income, and high income.

The variable “perceptions of corruption” in the dataset is continuous, with lower values indicating that fewer people perceive the government as corrupt. To conduct the chi-squared test, we need to categorize this continuous variable. The number of categories to create and the choice of thresholds can involve some subjectivity. For the sake of simplicity, we will divide the variable into two categories: “Not Corrupt” for countries with perceived corruption scores below the median and “Corrupt” for those with scores at or above the median. This approach allows us to examine the association between a country's income level and whether it is perceived as corrupt or not.

TABLE 4.2

Observed and expected frequencies of region and income group

Income:	Low	Lower Middle	Upper Middle	High
Not Corrupt	5 (5.5)	17 (17)	7 (14.5)	28 (20)
Corrupt	6 (5.5)	17 (17)	22 (14.5)	12 (20)

To perform the chi-squared test, we must first determine the absolute frequencies of the categories in the dataset. This can be achieved using a contingency table, which is a tabular representation of data that cross-classifies and summarizes the frequencies of observations based on two or more categorical variables.

Table 4.2 presents the observed frequencies for each combination of categories in the income group and perceived corruption variables. We can observe that among the 11 low-income countries, 5 are categorized as not corrupt while 6 are classified as corrupt. In the case of 34 lower-middle-income countries, half of them are perceived as corrupt, and the other half as not corrupt. Interestingly, 22 out of 29 upper-middle-income countries are perceived as corrupt, while this perception applies to only 12 out of the 40 high-income countries.

However, to determine whether there is a statistically significant association between income level and the perception of corruption, we must turn to hypothesis testing. The null hypothesis of the chi-squared test posits that there is no association between the categories of the two variables. We evaluate this null hypothesis by comparing the observed frequencies with the expected frequencies under the assumption that the null hypothesis is true. The expected frequencies, as indicated in Table 4.2, are calculated using software.

For instance, the expected frequency of “Not Corrupt” and “Corrupt” among upper-middle-income countries is both 14.5 due to the equal probability of being classified into either corruption category. Furthermore, the table reveals that there are fewer upper-middle-income countries classified as “Not Corrupt” than expected, and more high-income countries are perceived as “Not Corrupt” than anticipated. Notably, the observed and expected frequencies for lower middle-income countries in both corruption categories are identical.

The chi-squared test assesses the association between categorical variables by comparing observed and expected frequencies. It yields a chi-squared (χ^2) value as the test statistic. This statistic is computed by summing the squared differences between observed and expected frequencies within each cell of a contingency table, divided by the corresponding expected frequency. The chi-squared statistic provides an overall measure of the difference between observed and expected counts, rather than focusing solely on expected frequencies.

Since the chi-squared statistic follows the chi-squared distribution, software can be used to calculate the p-value based on this statistic and the degrees of freedom associated with the contingency table.

TABLE 4.3
Contribution two chi-squared statistic

Income:	Low	Lower Middle	Upper Middle	High
Not Corrupt	0.045	0	3.88	3.2
Corrupt	0.045	0	3.88	3.2

In the case of our contingency table examining income group and corruption, the chi-squared test yields a p-value of 0. This result leads to the rejection of the null hypothesis, indicating a likely association between the two variables. However, a statistically significant chi-squared test only suggests the probable existence of an association and does not specify the nature of the association or the categories involved.

To gain insights into the association between categories, we can compare observed and expected frequencies. In [Table 4.2](#), we observe that there are more upper-middle-income countries perceived as corrupt than expected, suggesting a potential association. Similarly, more high-income countries are perceived as not corrupt than anticipated, indicating a possible association. In contrast, the observed frequencies for lower middle-income and low-income countries closely match the expected values, suggesting a weak association.

To further validate these initial conclusions, we can assess the contribution of each cell in the contingency table to the chi-squared statistic, which quantifies the overall difference in the data.

[Table 4.3](#) reveals that upper-middle-income and high-income countries make the most significant contributions to the overall chi-squared, suggesting strong associations. On the other hand, lower middle-income and low-income countries make minimal contributions to the chi-squared statistic, indicating weak associations.

For the Python code utilized in this example, please refer to [Section 12.2.5](#) on p. 244.

4.4 The Non-parametric Approach

In the previous sections, we explored parametric approaches, which rely on predefined probability distributions and impose various requirements and assumptions on the data. In this section, we delve into the non-parametric approach, which does not depend on predefined probability distributions and avoids imposing specific requirements on the data.

One notable non-parametric technique is the bootstrap method. Like the idiom “pull yourself up by your bootstraps” suggests, the bootstrap method involves using the available data to generate new samples and estimate statistics

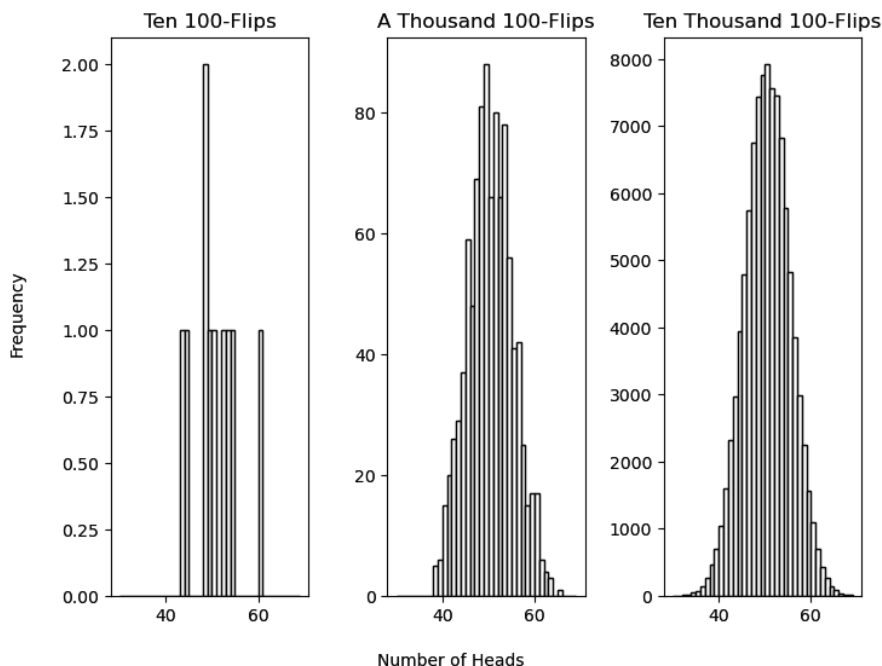
**FIGURE 4.5**

Illustration of the central limit theorem.

without relying on specific assumptions about the underlying data distribution.

Bootstrap makes use of the **central limit theorem**, which asserts that when a large number of random samples are drawn from a population, the means of those samples (referred to as the sampling distribution) will follow a normal distribution, irrespective of the shape of the population distribution. This theorem is particularly powerful in that it provides a foundation for the bootstrap method's effectiveness.

To demonstrate the central limit theorem, let's consider an example. Imagine we flip a balanced and fair coin 100 times and record the number of heads as one observation. This process is repeated to collect more observations. Since the outcome of flipping the coin is random, the number of heads per 100 flips will vary among observations. In [Figure 4.5](#), we can see histograms of heads with different sample sizes: 10, 100, 1,000, and 10,000 (each observation in the sample is generated by 100 coin flips). As the figures illustrate, the larger the sample size, the closer the distribution approximates a normal distribution.

To carry out the bootstrap method on a dataset, we follow several steps:

1. We start by drawing random samples from the original sample with replacement. These new samples are referred to as bootstrap

samplesbootstrap samples. It is crucial to note that the original sample must be representative of the population to ensure the validity of the bootstrap results. Random resampling of a representative sample allows us to simulate a distribution of population statistics under the null hypothesis.

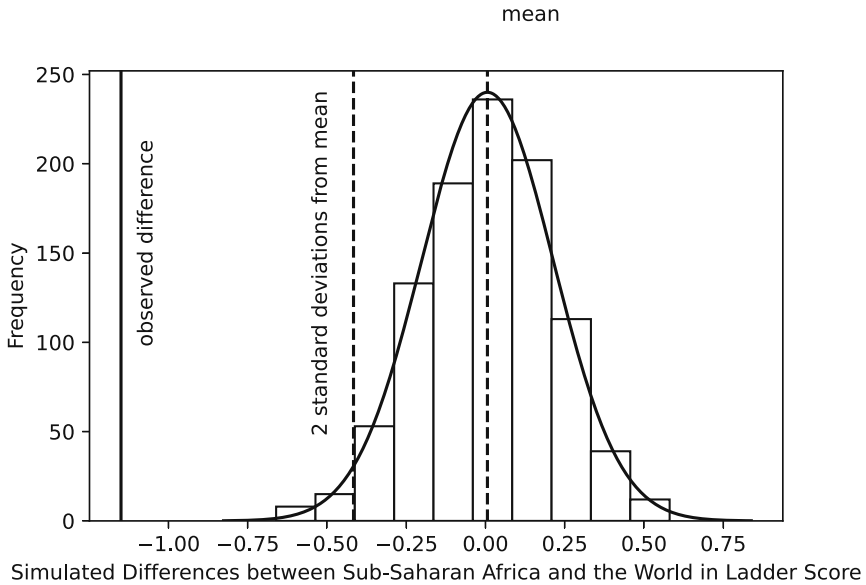
2. Next, we calculate the statistic of interest. Typically, this is the mean of the bootstrap sample, but it could also be the median or other statistics.
3. We then repeat the first two steps many times, often thousands of times. This process generates a distribution of statistics from bootstrap samples. According to the central limit theorem, this distribution should be close to a normal distribution and approximate the variation of the population statistics under the null hypothesis.
4. Finally, we compare the statistic from the observed data with the bootstrap distribution. This is where the concept of the p-value comes into play. If the statistic from the observed data is more extreme than 5% of the simulated samples, it means that the p-value is less than 0.05. In such cases, we claim to have enough evidence to reject the null hypothesis. Otherwise, if the observed statistic falls within the range of what is expected by random chance, we fail to reject the null hypothesis.

Now, let's revisit the Z-test that we discussed earlier, where we analyzed the difference between the mean Ladder scores of Sub-Saharan Africa and the world. In case we have doubts about the normal distribution assumption made in the test, we can perform a bootstrap analysis to validate the Z-test's conclusion. Since we want to compare the means of a sample (Sub-Saharan Africa) and the population (world), the test statistic could be the difference between the two means. In our observed data, the difference in mean Ladder scores between Sub-Saharan Africa and the world is -1.15. The question is whether this difference is statistically significant or simply a result of chance. The null hypothesis posits that there is no difference between the two means.

To initiate the analysis, we generate 1,000 bootstrap samples under the null hypothesis. While the choice of 1,000 is somewhat arbitrary, it is a commonly used number in practice.

Here is how the bootstrap samples are created:

1. In each bootstrap sample, Ladder scores are randomly resampled with replacement from the observed data, and they are reassigned among countries. This resampling process essentially constructs an "alternative universe" based on the null hypothesis. In this universe, Ladder scores are a product of random chance.
2. After generating each bootstrap sample, we calculate the difference between the mean ladder scores of Sub-Saharan Africa and the world, as it appears in that specific resampled dataset.

**FIGURE 4.6**

Sampling distribution of differences in means.

3. The first two steps are repeated 1,000 times to create 1,000 samples, each with its calculated difference in means. These 1,000 samples form a sampling distribution of the differences in means.

The resulting sampling distribution is depicted in [Figure 4.6](#). As we can see, it closely resembles a normal distribution, which aligns with the central limit theorem. The dotted line represents the observed difference that we calculated from the original data. This line is situated on the far left of the sampling distribution, indicating that if the null hypothesis is true (there is no actual difference between the two means), it is extremely improbable for us to obtain the observed data. This leads us to reject the null hypothesis, providing results consistent with what we found in the Z-test.

To see the code used for performing bootstrap, please navigate to [Section 12.3](#) on p. 245.

4.5 What's Next

In this chapter, we have delved into various hypothesis testing methods, exploring their applications, requirements, and interpretation. While these

methods provide valuable insights into the relationships between variables, they have their limitations. Most of them (except two-way ANOVA) are bivariate in nature, meaning they involve only two variables. Additionally, the methods either operate with a categorical or ordinal variable alongside a continuous or interval variable, as seen in the t-test and ANOVA, or require both variables to be non-continuous as demonstrated in the chi-squared test. Lastly, these tests primarily focus on revealing associations or differences between variables, leaving questions about the true impact of independent variables on the dependent variable largely unanswered.

In the next chapter, we will take a step further and explore regression analysis, a more flexible and powerful method that allows us to assess not only associations and differences between variables but also the impact of independent variables on the dependent variable. Specifically, we will dive into linear regression, learning how to interpret and evaluate regression results, unlocking a new dimension of data analysis. If you navigate to [Chapter 12](#), not only will you learn how to perform hypothesis testing in Python, but also you will learn how to automate repetitive tasks with loops.

4.6 Discussion Questions

1. Could you describe any real-world scenarios where hypothesis testing played a crucial role in decision-making, policy changes, or scientific discoveries?
2. How do you decide which hypothesis testing method is most appropriate for a given research question or dataset?
3. In what scenarios would you choose a one-tailed test over a two-tailed test, and vice versa? What are the implications of this choice?
4. What are the advantages and limitations of using parametric hypothesis tests compared to non-parametric tests?
5. When interpreting p-values, what is the significance of the 0.05 threshold?
6. What is the role of the central limit theorem in hypothesis testing?

Linear Regression Analysis

Regression analysis serves as a statistical method model relationships between a dependent variable and one or more independent variables. This approach finds wide applications in social science research, enabling hypothesis testing, causal effect estimation, and the exploration of association patterns. It empowers social scientists to address critical questions, such as:

- What is the impact of education on income?
- How do different types of political campaign advertisements influence voter behavior?
- What effects do social policies have on health outcomes?

In the context of regression analysis, the basic null hypothesis posits a lack of predictive connection between independent and dependent variables. Put simply, the null hypothesis implies that the coefficients measuring the influence of independent variables are zero.

Unlike the statistical tests discussed in the preceding chapter, regression analysis provides valuable insights into variable associations and the precise effects of independent variables on the dependent variable. A notable advantage of regression analysis is its adaptability to various levels of measurement. However, it is imperative to underscore that regression analysis, by itself, cannot establish causality. The establishment of causal relationships necessitates robust research designs, such as controlled experiments. Hence, careful consideration of causality is paramount when interpreting regression analysis results.

Regression techniques in data analysis fall into two principal categories: linear regression and non-linear regression, based on the underlying data patterns they model. They can also be grouped based on the number of dependent variables involved, resulting in simple regression (involving one dependent variable and one independent variable), multiple regression (involving one dependent variable and multiple independent variables), and multivariate regression (involving more than one dependent variable and one or more independent variables).

Each type of regression comes with its specific assumptions, advantages, and limitations. The choice of regression analysis type depends on the research question, data nature, and the theoretical framework under consideration. As an introductory exploration of regression analysis, this chapter will focus on

the Ordinary Least Squares (OLS) method, which forms the foundation for simple and multiple linear regressions, along with other statistical techniques. This foundational approach provides a fundamental grasp of regression analysis and establishes the basis for delving into more advanced regression techniques.

5.1 Simple Linear Regression

Simple linear regression is a method used to model the linear relationship between one dependent variable and one independent variable. To illustrate, consider our inquiry into whether the saying “money can’t buy happiness” holds true for countries, using the 2021 World Happiness Report data. In this analysis, we aim to understand the impact of income status on happiness. The dependent variable is happiness (measured as the Ladder score), while the independent variable is money (measured as logged GDP per capita). It is crucial to distinguish the roles of these variables in regression analysis; here, we are examining how money affects happiness.

A fundamental assumption of linear regression is that there exists a linear relationship between the dependent and independent variables. We can visually assess this linearity by creating a scatterplot that displays GDP on the horizontal axis and happiness on the vertical axis. In [Figure 5.1](#), we can observe the positive and approximately linear association between these variables, with a correlation coefficient of 0.79. These findings confirm that a linear regression model is appropriate for this analysis.

Identifying a linear association goes beyond mere visual inspection. We need a systematic approach to establish the optimal linear model. [Figure 5.2](#) presents three potential models that portray the relationship between the Ladder score and GDP. Among these, the solid line is created using the Ordinary Least Squares (OLS) method, while the dashed lines represent alternatives drawn based on arbitrary criteria (other possible lines that obviously do not fit the data are purposefully excluded from illustration). Despite the visual appeal of the dashed lines in capturing the pattern, it is the OLS-fitted line that most accurately represents the underlying relationship.

To determine this OLS line, which is the best among various alternatives, and to assess its efficacy as a linear model, we will go through three key steps:

1. Identify the OLS line.
2. Test its statistical significance using the p-value.
3. Evaluate the model’s explanatory power with R-squared.

While modern computer software can perform these calculations automatically (as discussed in [Chapter 13](#)), it is valuable to understand how and why they are computed.

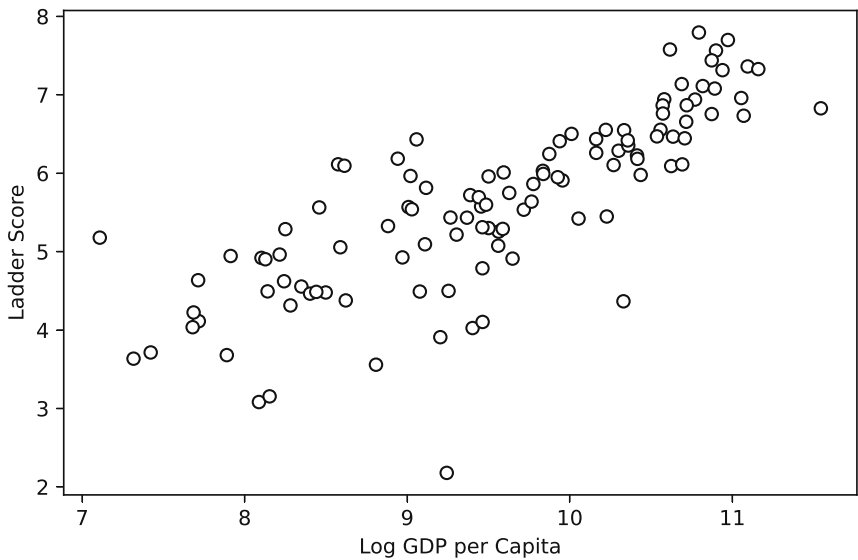


FIGURE 5.1
Relationship between happiness and economic development.

5.1.1 The Ordinary Least Squares (OLS) Method

Starting with a concise delve into algebra, we can represent a straight line on a 2D plane through the equation $y = a + bx$. In this equation, a corresponds to the intercept, b represents the slope, and both y and x are variables. This equation precisely encapsulates a simple bivariate linear regression model, visually depicted as a straight line. In the realm of regression analysis, y symbolizes the dependent variable, and x is the independent variable. The slope, denoted as b , quantifies the extent to which changes in the independent variable influence the dependent variable, whereas the intercept, a , signifies the projected value of y when x equals zero.

Let us explore examples of linear relationships. Imagine embarking on a road trip, cruising at a constant speed of 60 miles per hour. The relationship between the trip’s duration and distance is linear ($Distance = 60m/h \times Duration + 0$). In this model, distance serves as the dependent variable, while duration acts as the independent variable. The slope, in this case, is 60, signifying that each one-hour increase in duration corresponds to an additional 60 miles in distance. The intercept, represented as 0, implies that when you spend 0 hours driving, you cover a distance of 0 miles.

Suppose we have three observed values for the “duration” variable and seek to employ the linear model to estimate the respective “distance” values. [Figure 5.3](#) provides a visual representation of the linear model alongside the

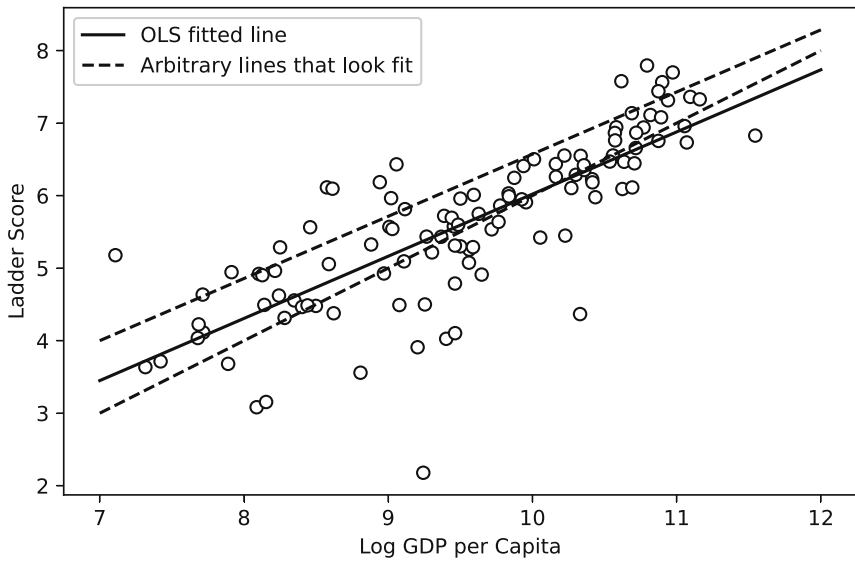


FIGURE 5.2
Identify the best fitted linear model.

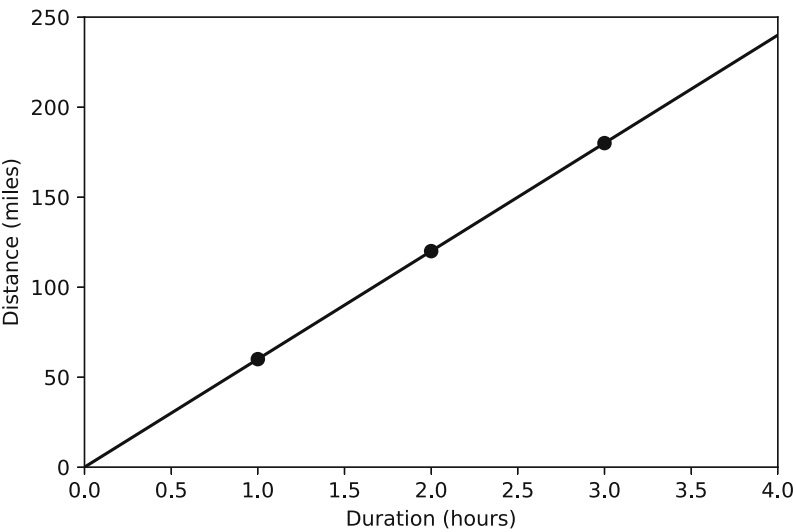


FIGURE 5.3
Linear model for inches and centimeters.

three data points. It is important to note that each point on the line represents an estimated distance value (on the vertical axis) based on a specific duration value (on the horizontal axis). The fact that all three data points align with the line demonstrates that the model effectively utilizes duration to accurately estimate distance.

While it is an ideal scenario to have a model that can perfectly predict the dependent variable, the real world often involves a degree of error in predictions. Perfect models are rare, highlighting the importance of meticulous interpretation and result validation. Let’s revisit the relationship between the Ladder score and GDP, as depicted in Figure 5.2. This time, we will focus on Zambia, Albania, and Finland as examples, presented in Figure 5.4 for a more compact view.

As evident in Figure 5.4, the OLS fitted line does not intersect with any observed data point. This indicates that the linear model represented by the line does not accurately estimate any of the observations. Specifically, the model overestimates the Ladder scores for Zambia and Albania but underestimates the Ladder score for Finland.

In regression analysis, the difference between observed and estimated values of the dependent variable is termed residuals (as represented by the dashed vertical lines in Figure 5.4). If the observed value is greater than the estimated value, the residual is positive. Conversely, if the observed value is smaller than

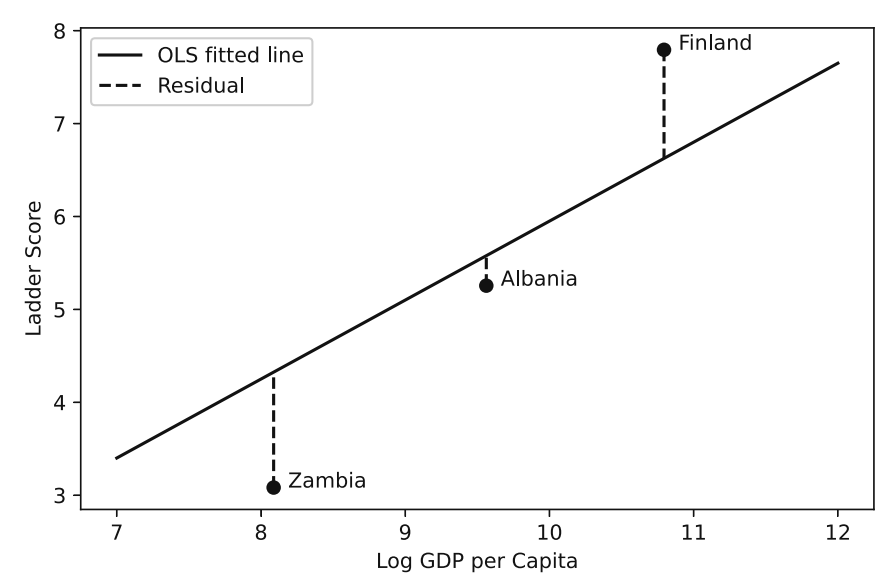


FIGURE 5.4
Residuals of linear regression.

the estimated one, the residual is negative. When the two values are equal, the residual is zero.

Since a residual indicates how much the model estimate deviates from the actual observation, the sum of squared residuals is used to assess the model's goodness-of-fit. This approach prevents positive and negative residuals from offsetting each other, which could potentially yield a deceptively low or zero residual. Consequently, the smaller the sum of squared residuals is, the better the model fits the observed data.

The OLS method determines the best-fitting line's slope and intercept by *minimizing* the sum of squared residuals. Of these estimated parameters, the slope (denoted as b) holds more significance as it reveals the relationship between the independent and dependent variables. The interpretation of the slope is as follows: for each one-unit change in the independent variable, the dependent variable undergoes a change of b units.

It is important to note that the OLS method solely provides a line that best fits the data, given the variables or data at hand. However, it does not indicate whether the observed relationship (the slope) is statistically significant enough to reject the null hypothesis.

Statistical Significance

To assess the statistical significance of the slope coefficient, it is crucial to determine its p-value. To calculate this p-value, a process similar to the tests discussed in the previous chapter is used. The estimated slope coefficient is divided by its standard error, producing a t-statistic. This t-statistic measures the extent of deviation between the observed and expected slope coefficients under the null hypothesis, which assumes a value of 0. The p-value can then be computed using this t-statistic and the degrees of freedom, which for a simple regression model, equate to the number of observations minus the number of estimated parameters (two in this case: slope and intercept).

If the p-value of the slope coefficient is less than or equal to 0.05, according to conventional standards, it is considered statistically significant, leading to the rejection of the null hypothesis. On the other hand, if the p-value exceeds 0.05, the null hypothesis is not rejected.

While a fitted line and a statistically significant slope coefficient provide insights into the impact of the independent variable on the dependent variable, they do not fully elucidate the model's effectiveness in explaining and estimating the dependent variable (or, explanatory power of the model). To address this, another measure of goodness-of-fit is required: the coefficient of determination, commonly known as R-squared.

Evaluate Explanatory Power of the Model: R-squared

The R-squared (or, R^2) value, often referred to as the coefficient of determination, is a key metric used to assess a model's goodness-of-fit by focusing on how well it explains the variation in the dependent variable. A strong

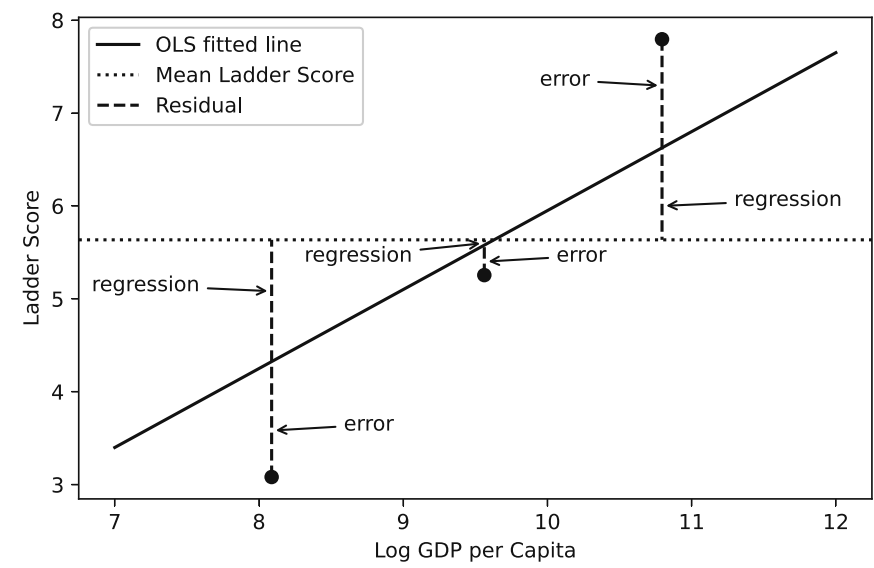


FIGURE 5.5
Explanation for variation in the dependent variable.

model should not only elucidate the relationship between variables but also accurately estimate outcomes.

R-squared quantifies the extent of variation in the dependent variable and is calculated by examining the sum of squares of the differences between its observed values and its mean (referred to as the total sum of squares, SST). Although this might seem similar to the OLS formula, there is a subtle difference. OLS concentrates on the sum of squared residuals, which represent the differences between observed and estimated values of the dependent variable. On the other hand, R-squared assesses the difference between the observed value and the mean of the dependent variable.

Figure 5.5 utilizes the data from Figure 5.4 and illustrates the deviations of the observed Ladder scores from the mean Ladder scores. Each deviation consists of two components: regression and error. The regression component reflects the amount of variation that the OLS model can account for. The remaining deviation, unexplained by the model, is attributed to errors stemming from variables we opt to exclude from the model or latent variables that cannot be directly observed or measured.

The closer the observed values align with the fitted line, the more effectively the regression model can explain variations in the dependent variable.

To manually calculate the R-squared value (indicating the proportion of variance explained by the model), we divide the regression sum of squares (SSR) by the total sum of squares (SST) and then subtract the result from

TABLE 5.1
Regression results

	OLS Coefficients
Log GDP per capita	0.857*** (-0.063)
constant	-2.55*** (0.6)
N	111
R ²	0.628

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

1. Additionally, the R-squared value also equals the square of the correlation coefficient between the dependent and independent variables.

R-squared ranges between 0 and 1. A higher R-squared value indicates that the model can explain a greater portion of the variation in the dependent variable, signifying a more robust model. A linear regression model with an R-squared of 1 implies that it can explain 100% of the variations in the dependent variable. For example, the previous model that estimates distance with travel duration has an R-squared of 1, indicating it explains all variations in distance based on duration.

5.1.2 Example: Impact of GDP on Ladder Score

Let’s now apply the OLS regression to investigate the influence of log GDP per capita on a country’s happiness. Table 5.1 provides an overview of the OLS regression results (as reported in academic publications), comprising three sections.

The first section reports the OLS regression coefficients. According to our OLS analysis (code in Section 13.2.1 on p. 257), the slope coefficient for log GDP per capita is 0.857. It is customary to include the standard error of a coefficient in parentheses beneath the coefficient, which serves as a crucial component in determining the p-value.

Typically, researchers represent the p-value using asterisks (*). The number of asterisks corresponds to the level of significance, with more asterisks indicating a lower p-value. However, it is important to note that different fields may adhere to varying conventions for p-values and their representation in results tables. For instance, some disciplines consider p-values below 0.05 as statistically significant. In these cases, one asterisk might signify a p-value of 0.05, two asterisks 0.01, and three asterisks 0.001. Additionally, researchers sometimes employ different symbols (e.g., +) to denote various significance levels. Therefore, it is crucial to transparently report the chosen symbol or p-value threshold in the table (conventionally, below the table).

The result reveals that the coefficient of log GDP per capita is statistically significant. This indicates that a one-unit change in log GDP per capita is estimated to lead to a 0.857-unit change in the Ladder score.

In cases where a variable is not statistically significant, we fail to reject the null hypothesis, suggesting that the variable may have no discernible relationship with the Ladder score. As for the constant coefficient, while it is statistically significant, it does not impact our hypothesis testing because it represents a constant term, serving as an intercept. However, the constant coefficient remains essential for estimating a country's Ladder score based on its log GDP per capita. Mathematically, we can express the estimated simple linear regression model as follows: $Ladder\ Score = 0.857 \times Log\ GDP\ per\ Capita - 2.55$. This model can be utilized to make new predictions. For example, in 2021, China's log GDP per capita stood at 9.776. Based on the OLS model, its estimated Ladder score would be 5.847 ($9.776 \times 0.857 - 2.55$). Notably, China's observed Ladder score in 2021 was 5.862, which closely aligns with our estimation!

The second section of the table describes the overall model and its performance. The "N" stands for the sample size. In the example dataset, there are 111 countries. You may notice that the sample size is smaller than the number of countries in the dataset. This is because countries with missing values are excluded from regression analysis. The " R^2 " represents the R-squared value. The " R^2 " reported in the table indicates that the model explains 62.8% of the variations of the dependent variable, which is quite good for a model with only one independent variable.

5.1.3 Assumptions of the OLS simple regression

There are crucial assumptions that must be met for the OLS regression to provide valid and reliable results. Ensuring these assumptions are satisfied is essential, as the quality of our model is heavily influenced by the quality of the input data. Remember the saying: "Garbage in, garbage out." Let's explore these assumptions:

Assumption 1: Linearity. There should be a linear relationship between the independent and dependent variables. Before applying a linear model, it is vital to visually assess this linearity. You can do this by examining a scatterplot of the variables. This initial step helps confirm the presence of a rough linear relationship.

Assumption 2: Independence of observations. For the model to perform effectively, the observations within the sample should be independent of one another. Independence can often be achieved through random sampling. This assumption is critical to avoid issues related to dependent or correlated observations.

Assumption 3: Homoscedasticity of Residuals. Homoscedasticity refers to the idea that residuals (differences between observed values and the model's estimated values) should have consistent variability across all levels of the independent variable (or dependent variable). In simpler terms, the

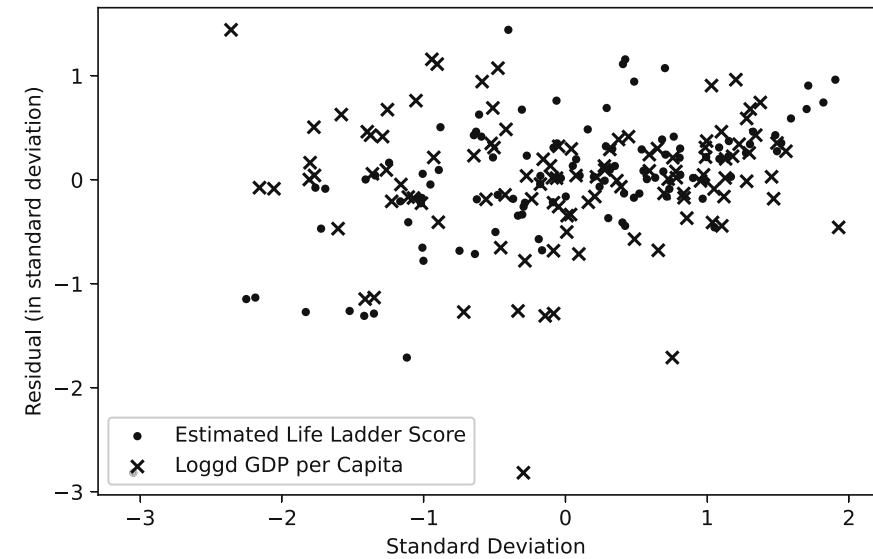


FIGURE 5.6
Testing homoscedasticity assumption.

differences between observed values and the model’s estimations should exhibit roughly the same amount of variation throughout the entire range of the independent variable. If the residuals exhibit varying degrees of variability, we refer to them as heteroscedastic. A reliable model should allow us to make estimations without concerns about some estimations having small errors while others have large errors.

An effective way to examine if the homoscedasticity assumption holds for the model is through visual inspection. You can achieve this by creating a scatterplot that displays the residuals in relation to the independent variable (or dependent variable, when there are multiple independent variables). [Figure 5.6](#) presents a visual test that incorporates both the independent variable (logged GDP per capita) and the estimated dependent variable (Life Ladder score). To facilitate comparison, the data is standardized since the two variables are not measured on the same scale.

In this scatterplot, we can observe that the vertical spread of residuals along the horizontal axis remains mostly stable, fluctuating within a range of -1 to 1. This visual consistency signifies homoscedasticity, which is a desirable characteristic in regression analysis. For contrast, [Figure 5.7](#) showcases residuals that exhibit heteroscedastic behavior. In this case, the variability of the residuals changes along the horizontal axis, which is not ideal.

While visual inspection is a straightforward method for assessing homoscedasticity, there are also statistical tests available for this purpose. However, the details of these tests are beyond the scope of this book.

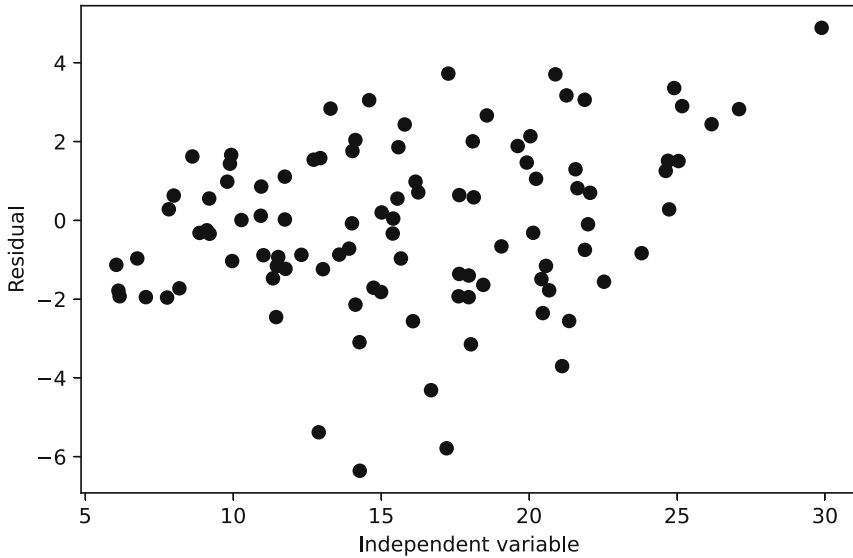


FIGURE 5.7
Heteroscedastic residuals.

Assumption 4: Normality of residuals. The residuals should exhibit a normal distribution with an approximate mean of zero. The normality assumption entails that most residuals generated by our model cluster around 0. In [Figure 5.8](#), we can observe a histogram representing the residuals produced by our OLS model. With the exception of one residual having an extreme value (-3.12), the overall distribution illustrates that the majority of residuals are in close proximity to 0. For a more formal assessment of normality, you could employ a statistical test such as the Shapiro–Wilk normality test on the residuals (refer to p. 65).

These assumptions are critical in ensuring the OLS method generates accurate and meaningful results. When these assumptions are met, we can have confidence in the validity and reliability of the model’s output. In practice, it is essential to evaluate and validate these assumptions before relying on the OLS regression results.

It is important to note that testing for homoscedasticity and normality can only be performed after creating and executing the regression analysis. If our model happens to violate either of these assumptions, it does not necessarily warrant discarding the model entirely, as outliers within the data might be contributing to these deviations.

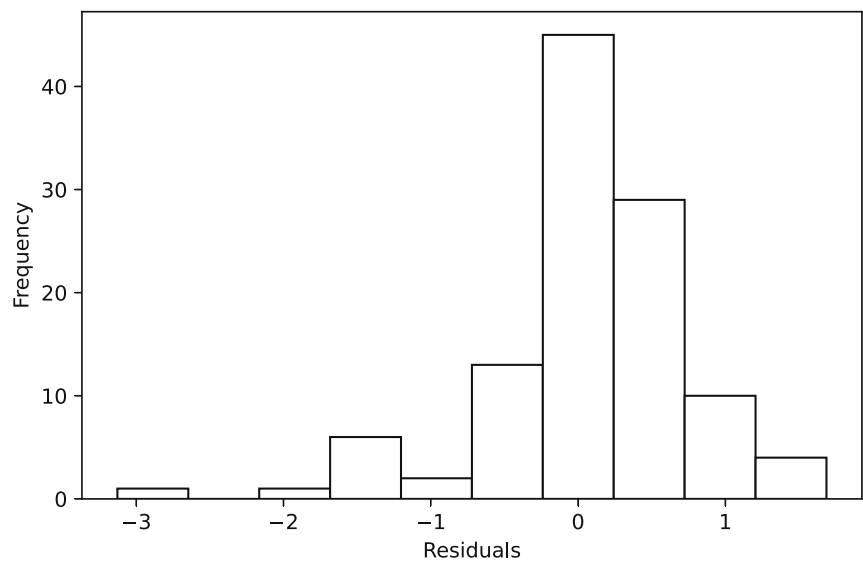


FIGURE 5.8
Testing normality of residuals.

5.1.4 Impact of Outliers on Regression Results

As discussed in [Section 2.3.5](#), the decision to remove outliers is a subject of debate. To refine the model’s fit, we proceed to identify and exclude outliers from both the dependent and independent variables. We employ a 1.5 times the interquartile range threshold to identify outliers. This approach involves calculating the difference between the 25th and 75th percentile values the interquartile range ([Section 11.4](#) on p. 226 covers how to find these values), and then multiplying this difference by 1.5. Any observation with a value exceeding the 75th percentile value plus 1.5 times the interquartile range or falling below the 25th percentile value minus 1.5 times the interquartile range is considered an outlier.

This process identifies Afghanistan and Lebanon as outliers in the Ladder score dataset.

The updated regression results are presented in [Table 5.2](#). Notably, the independent variable remains statistically significant, although its coefficient is marginally lower than that in the original model. Furthermore, the removal of outliers has led to an improved R^2 value, increasing from 0.628 to 0.674. For an evaluation of the assumptions of homoscedasticity and normality, please refer to [Figures 5.9](#) and [5.10](#). These revised figures indicate a more consistent variability and spread of residuals compared to the original figures, signifying enhancements in homoscedasticity and normality of the residuals.

TABLE 5.2
Regression results (without outliers)

	OLS Coefficients
Log GDP per capita	0.849*** (0.057)
constant	-2.44*** (0.546)
N	110
R ²	0.674

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

However, it is crucial to exercise caution when handling outliers. By removing outliers, we effectively select observations that “fit” our model while excluding others that do not. This approach could potentially introduce bias into our analysis.

5.2 Linear Multiple Regression

Linear multiple regression investigates the relationship between a dependent variable and multiple independent variables. This approach employs multiple independent variables to elucidate their linear relationship with the dependent variable, expressed mathematically as: $y = a + b_1x_1 + b_2x_2 + \dots + b_nx_n$, where x_n represents the n th variable and b_n refers to the slope coefficient of the n th variable.

It is essential to distinguish multiple regression from multivariate regression, which pertains to scenarios with multiple dependent variables. The focus of this book remains on multiple regression.

Incorporating numerous independent variables into a regression model enhances its capacity to estimate the dependent variable and expound its variations. However, it is crucial to exercise discretion when deciding which variables to include. The selection should be theoretically grounded, aligned with the research framework, and hypothesis-driven. Variables should be included if they are deemed relevant based on the underlying theory.

From a methodological perspective, multiple regression introduces the assumption of lack of multicollinearity, in addition to all the assumptions of simple OLS regression. Multicollinearity arises when multiple independent variables in a regression model exhibit high correlations. This phenomenon is not uncommon in social science studies. When highly correlated independent variables are statistically significant, it becomes challenging to discern which variable truly impacts the dependent variable. In cases of multicollinearity, we

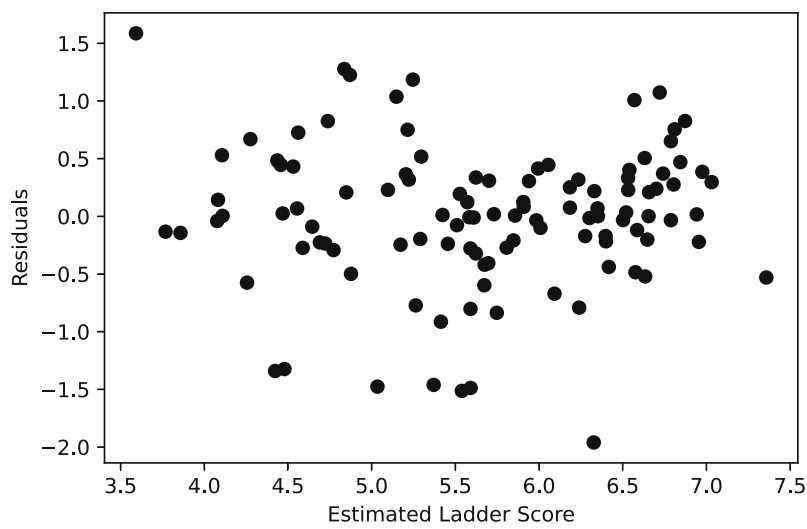


FIGURE 5.9
Homoscedasticity test (without outliers).

can retain one of the correlated variables or apply variable transformations to mitigate the correlation.

Table 5.3 presents the correlation coefficients among the continuous variables in the World Happiness Report dataset. Each cell in the table represents the correlation coefficient between the variables listed in the corresponding row and column. The numbers in the column headers correspond to the variables named in the first column.

Among the three variables, let us choose to retain log GDP per capita, with the selection based on theoretical considerations. Table 5.4 presents the

TABLE 5.3
Correlation table to test multicollinearity

	1	2	3	4	5	6	7
1 Log GDP per capita	1.00	–	–	–	–	–	–
2 Social Support	0.75	1.00	–	–	–	–	–
3 Life Expectancy	0.86	0.72	1.00	–	–	–	–
4 Freedom	0.33	0.47	0.38	1.00	–	–	–
5 Generosity	0.00	0.15	-0.01	0.21	1.00	–	–
6 Corruption	-0.39	-0.26	-0.35	-0.40	-0.19	1.00	–
7 Confidence in Gov.	-0.21	-0.17	-0.16	0.33	0.19	-0.55	1.00

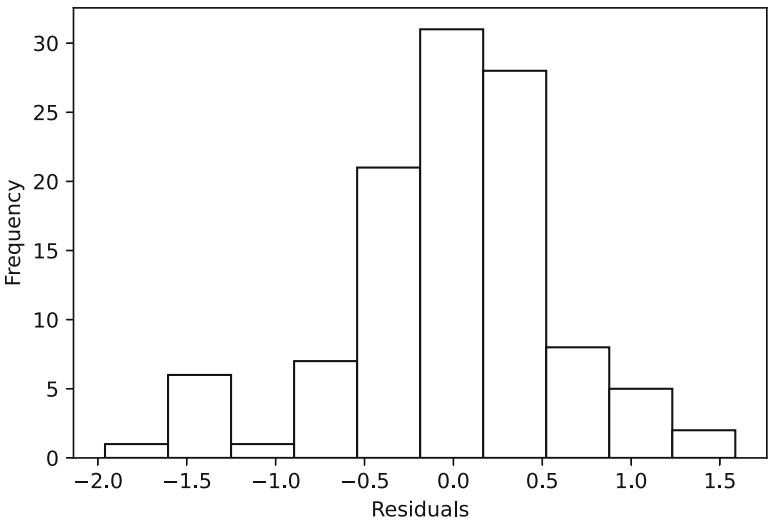


FIGURE 5.10
Normality test (without outliers).

TABLE 5.4
Multiple regression results

	OLS Coefficients
Log GDP per capita	0.63*** (0.069)
Freedom	3.65*** (0.626)
Generosity	0.43 (0.361)
Corruption	-1.03** (0.441)
Confidence in gov.	-1.19** (0.421)
constant	-1.93** (1.086)
N	99
R ²	0.788

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

outcomes of the multiple linear regression. When comparing the multiple regression results to the simple regression presented earlier, several observations can be made:

1. Multiple regression incorporates additional independent variables and their corresponding coefficients.
2. In the multiple regression, the coefficient for log GDP per capita (0.63) is lower than that in the simple regression (0.849). This difference can be attributed to other variables in the multiple regression sharing the responsibility for explaining variations in the Ladder score.
3. Generosity is not statistically significant among the variables, while the p-value for the perception of corruption is below 0.1.
4. The multiple regression model boasts a higher R-squared value (0.788) compared to the simple regression model, thanks to the enhanced explanatory power resulting from the additional independent variables.
5. The sample size is reduced due to missing values in some of the variables.

Interpreting the slope coefficients in multiple regression is akin to the process in simple regression, but with a caveat. The estimates of the slope coefficients in multiple regression assume that all other independent variables remain constant (also referred to as *ceteris paribus* in Latin). For instance, based on the regression results, *ceteris paribus*, a one-unit increase in log GDP per capita corresponds to a 0.63-unit increase in the Ladder score. In the case of confidence in the government, the statistically significant and negative coefficient indicates that, *ceteris paribus*, a one-unit increase in confidence results in a 1.19-unit decrease in the Ladder score. Concerning generosity, as it is not statistically significant, there is no evidence to suggest that it affects the Ladder score, despite the presence of a slope coefficient in the regression model.

In the mathematical expression of the model, only statistically significant variables are included. Therefore, the results in [Table 5.4](#) are expressed as follows: $Ladder\ Score = 0.63 \times Log\ GDP\ per\ Capita + 3.65 \times Freedom - 1.03 \times Corruption - 1.19 \times Conf\ in\ Government - 1.93$. This equation provides a way to estimate the Ladder score based on the values of the included independent variables.

5.2.1 Include Ordinal and Nominal Variables in Regression

Incorporating ordinal and nominal variables into regression models requires specific handling. For example, let us replace log GDP per capita with the income group variable (with categories from low income to high income) that we created previously. We need to first convert the income group variable from

string data type into numeric values (e.g., 1, 2, 3, or 4), with 1 representing the lowest level and 4 the highest. Then, we add it to the regression model as we would with a continuous variable. Keep in mind that although we can include ordinal variables in this way, their range of values is limited compared to true continuous variables. In the case of the income group variable, the range of possible changes is limited to values between 1 and 4.

Adding a nominal variable to a regression model is a bit different. For instance, you might want to include the “region” variable from the World Happiness Report data to explain variations in the Ladder score. The “region” variable consists of seven unique regions. As discussed in [Section 2.3.1](#), nominal variables do not have numeric meanings like continuous or ordinal variables. Therefore, regression cannot estimate changes in the dependent variable for each unit increase or decrease in the independent variable, as it can with continuous or ordinal variables.

To incorporate a nominal variable into a regression model, you can convert its categories into multiple dummy variables. Dummy variables typically have two values, such as 1 and 0, representing “yes” and “no” or “True” and “False.” This conversion can be done manually or using software (see [Section 13.2.4](#) on p. 262 for code example). The regression model then considers whether a country belongs to a specific region (1) or not (0), comparing countries in one region with those in other regions.

[Table 5.5](#) updates the regression results by adding the “region” variable as a nominal variable. You may notice that “East Asia and Pacific” is missing in the regional categories while there are seven regions in the dataset. This is because there is no need to convert all categories into dummy variables. Each region is already represented as a dummy variable. When a country does not belong to any of the listed six regions and there are no missing values, it must belong to the “East Asia and Pacific” region. Additionally, the regression uses the “missing” category as the baseline for comparison with other categories.

The results presented in the table show that the only statistically significant coefficient is for the “South Asia” region, with a coefficient of -0.95. This indicates that, under identical conditions, a country in South Asia is estimated to have a Ladder score 0.95 units lower than that of a comparable country in East Asia and Pacific (baseline category). The lack of statistical significance for coefficients associated with other regions suggests that we do not have sufficient evidence to claim the existence of differences between these regions and East Asia in terms of Ladder scores, given the same conditions for other variables.

In contrast to the findings in [Table 5.4](#), there is a notable difference in the “confidence in government” variable. While it was statistically significant previously, it has now become insignificant. The diminished significance suggests that while confidence in government may have previously contributed to explaining variations in the Ladder score independently of the regional variable, its explanatory power diminishes in the presence of the regional variable. This phenomenon implies that there may be associations between confidence

TABLE 5.5
Results with categorical variables in regression

	Coefficients
Log GDP per capita	0.64*** (0.094)
Freedom	3.32*** (0.706)
Generosity	0.61 (0.389)
Corruption	-0.81* (0.43)
Confidence in Gov.	-0.73 (0.442)
Region.Europe & Central Asia	0.17 (0.195)
Region.Latin America & Caribbean	0.32 (0.233)
Region.Middle East & North Africa	-0.04 (0.301)
Region.North America	0.07 (0.4)
Region.South Asia	-0.95*** (0.356)
Region.Sub-Saharan Africa	0.09 (0.262)
constant	-2.23* (1.319)
N	99
R ²	0.821

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

in government and regions, with regions being a more compelling explanatory factor than confidence in government. This suspicion is supported by an improvement in the R² value, which has increased from 0.788 to 0.821, indicating a better fit for the model.

5.2.2 Include an Interaction Term in Regression

So far, we have focused on the relationship between a dependent variable and a single independent variable in isolation from other variables in the model. This approach assumes that the relationship between these two variables remains consistent across different levels of other factors. However, in practice, we often need to hypothesize that the relationship between two variables may be influenced by the presence of a third variable. In such cases, we introduce this third variable as an interaction term in our regression model.

TABLE 5.6
Regression results with interaction term
between two continuous variables

	Model 1	Model 2
Corruption	-0.61*** (0.08)	-2.12** (0.67)
Social Support	-0.5*** (0.17)	-1.8** (0.58)
Corruption × Support		1.7** (0.75)
constant	1.33*** (0.12)	2.48*** (0.52)
N	108	108
R ²	0.408	0.436

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

In a regression model with an interaction term involving two independent variables, the mathematical representation is as follows: $y = a + b_1x_1 + b_2x_2 + b_3(x_1x_2)$, where x_1 and x_2 represent the two independent variables, and the product of the two independent variables, x_1x_2 , is the interaction term. Coefficients b_1 and b_2 corresponds to the main effect of the independent variables, while b_3 is the coefficient associated with the interaction term, known as the interaction effect.

Interaction between Two Continuous Variables

Let us look into the relationship between people’s confidence in their government and two continuous variables: perception of corruption and social support, which measures the significance of social connections and relationships in people’s happiness. Table 5.6 presents the findings from two regression models. Model 1 showcases a multiple regression without an interaction between corruption and social support, while Model 2 introduces an interaction term between them.

To initiate the discussion on interaction, let us direct our attention to the coefficient of corruption as reported in Model 1. According to this model, assuming social support remains constant, a one-unit increase in people’s perception of corruption equates to a 0.61-unit decrease in confidence in government. This relationship is mathematically expressed as *Conf In Government* = $-0.61 \times Corruption - 0.5 \times Social\ Support + 1.33$.

To estimate confidence in government concerning corruption levels, we can conduct an experiment by setting various values for social support. Suppose we select two levels, 0.7 and 0.9, positioned around the 25th and 75th percentiles of the variable, respectively. With a social support value of 0.7, the estimated confidence in government is $-0.61 \times Corruption + 0.98$. For a

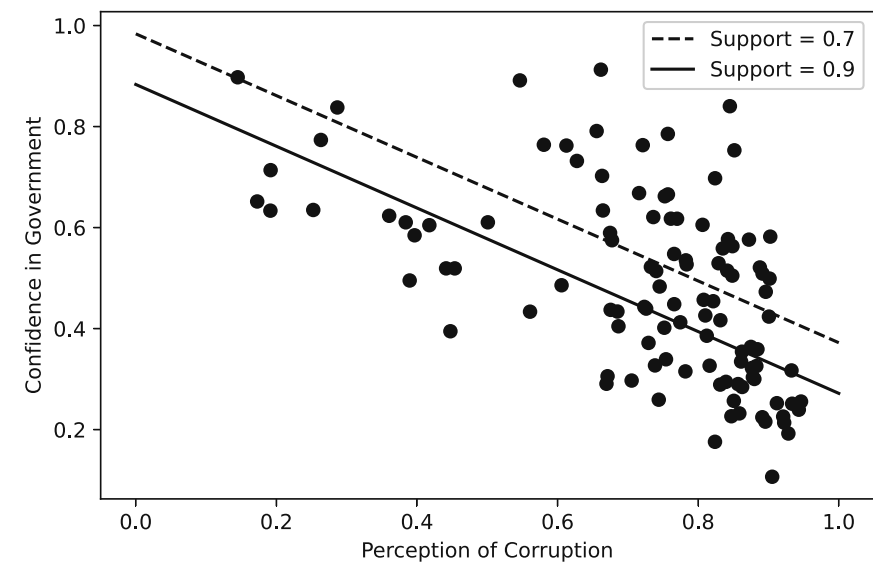


FIGURE 5.11
Parallel slope assumption in the relationship between corruption and confidence in government.

social support value of 0.9, the estimated confidence in government becomes $-0.61 \times Corruption + 0.88$.

Figure 5.11 depicts the relationship between corruption and confidence in government at two levels of social support. The dashed line represents the impact of corruption on confidence in government with social support at 0.7, while the solid line portrays this relationship when social support is at 0.9. Both lines remain parallel because the slope coefficient of corruption remains constant. Any changes in social support values solely influence the intercept, which determines the vertical position of the lines.

The consistent parallel behavior is a result of the parallel slope assumption in multiple regressions. This assumption posits that the corruption coefficient remains constant, unaffected by other independent variables. This signifies that the way an independent variable is associated with the dependent variable does not alter based on the values of other independent variables.

Whether the data meets the parallel slope assumption can be investigated with the interaction term. When data violates the assumption, the relationship between an independent variable and the dependent variable changes as a third variable changes.

Model 2 in Table 5.6 introduces an interaction term between the independent variables, revealing differences in the coefficients compared to Model 1.

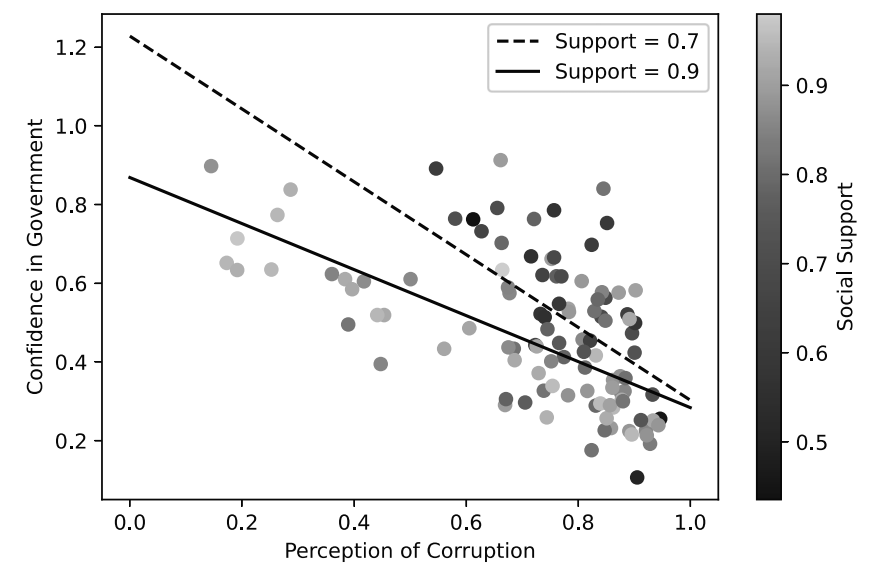


FIGURE 5.12
Effect of interaction in the relationship between corruption and confidence in government.

While all coefficients remain statistically significant, their values differ, showcasing the influence of the interaction term.

Mathematically expressed as $Conf\ in\ Government = -2.12 \times Corruption - 1.8 \times Social\ Support + 1.7 \times Corruption \times Social\ Support + 2.48$, the impact of corruption on confidence in government changes with varying levels of social support. Estimations with social support values of 0.7 and 0.9 yield $Conf\ in\ Government = -0.93 \times Corruption + 1.22$ and $Conf\ in\ Government = -0.59 \times Corruption + 0.86$, respectively.

Figure 5.12 offers a visual representation of these variations. The dark and light markers represent high and low social support values. The figure demonstrates that observations with high social support levels are more concentrated on the left and lower half of the distribution, whereas those with low social support levels are more concentrated on the right and upper half. The distinct clusters indicate that the relationship between corruption and confidence in government varies based on levels of social support, as depicted by the changes in the coefficient values.

TABLE 5.7
Regression results with interaction term between corruption and region

	Model 3	Model 4
Corruption	-0.61*** (0.07)	-0.45** (0.16)
Region.Europe & Central Asia	0.001 (0.05)	0.04 (0.12)
Region.Latin America & Carriibbean	-0.02 (0.05)	0.65** (0.32)
Region.Middle East & North Africa	0.11* (0.06)	0.96** (0.32)
Region.North America	-0.05 (0.11)	0.09 (0.35)
Region.South Asia	0.17** (0.08)	1.8** (0.72)
Region.Sub-Saharan Africa	0.18*** (0.05)	0.56** (0.25)
Region.Europe & Central Asia × Corruption		-0.07 (0.18)
Region.Latin America & Carriibbean × Corruption		-0.9** (0.41)
Region.Middle East & North Africa × Corruption		-1.13** (0.42)
Region.North America × Corruption		-0.23 (0.63)
Region.South Asia × Corruption		-2.01** (0.87)
Region.Sub-Saharan Africa × Corruption		-0.53 (0.33)
constant	0.88*** (0.06)	0.79*** (0.1)
N	108	108
R ²	0.482	0.508

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

Interaction between a Continuous Variable and a Categorical Variable

An interaction term can be between continuous and categorical variables. In the following example, let us examine the impact of perception of corruption (continuous) and region (categorical) on confidence in government. [Table 5.7](#) presents the results of two OLS models. Model 3 is a linear multiple regression with corruption and region as independent variables. Model 4 adds an interaction term between corruption and region. Since region is a categorical variable converted into multiple dummy variables of specific regions, the interaction

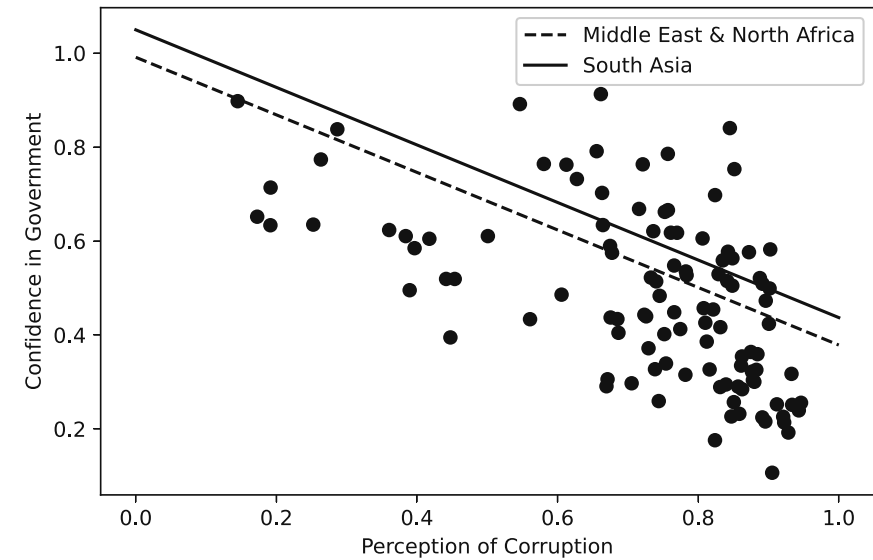


FIGURE 5.13
Parallel slopes of regions.

term in the model turns into multiple pairs of products between corruption and a specific region.

Model 3 can be mathematically represented as $Conf\ in\ Government = -0.61 \times Corruption + 0.11 \times Region.Middle\ East\ \&\ North\ Africa + 0.17 \times Region.South\ Asia + 0.18 \times Region.Sub-Saharan\ Africa + 0.88$. Each of the regions is now converted to a dummy variable and can take only 0 or 1 as a value, with 1 being positive and 0 being negative of the condition. These dummy variables serve to differentiate the relationships among distinct regions. For example, to estimate the confidence in government of a country in South Asia, the dummy variable is 1 for South Asia is 1 and 0 for other regions. The mathematical formula for South Asia becomes $Conf\ in\ Government = -0.61 \times Corruption + 0.17 \times 1 + 0.88$.

Figure 5.13 illustrates the relationships between corruption and confidence in government for two specific regions: South Asia, and the Middle East and North Africa. The plotted lines represent the parallel slopes between these regions, highlighting consistent relationships within the model that lacks interaction terms.

Conversely, Model 4 indicates statistical significance in “corruption,” “Middle East and North Africa region,” and “South Asia region,” as well as their interaction terms. This deviation from the parallel slope assumption suggests different relationships as a consequence of the interaction terms. Figure 5.14 visually distinguishes countries from these regions, showcasing their unique

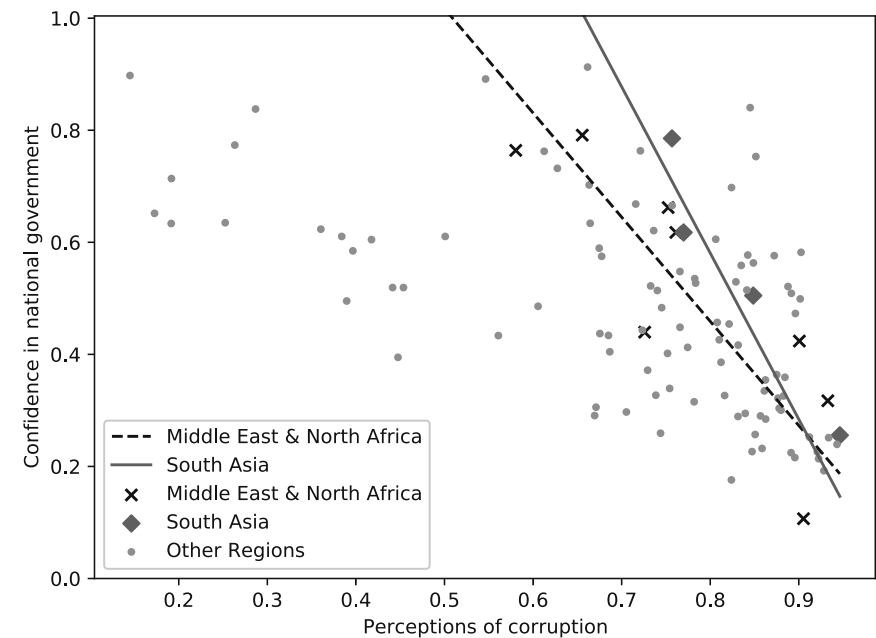


FIGURE 5.14
Interaction between region and corruption.

distribution patterns compared to other countries. Notably, the steeper slope for South Asian countries, compared to the Middle Eastern and North African countries, signifies a more pronounced impact of corruption on confidence in government in the former.

When Main Factors and Their Interaction Differ in Significance

Table 5.7 reveals scenarios where regions or their interaction terms exhibit varied statistical significance. In cases where the main factors are individually significant, yet their interaction term is not, exemplified by Sub-Saharan Africa region, it signifies an absence of interaction effect between the main factors. This suggests that being situated in Sub-Saharan Africa does not alter the influence of corruption on confidence in government.

Conversely, instances may occur where the main factors lack significance, yet their interaction term is significant. This occurrence denotes that the individual main factors alone do not substantially influence the dependent variable. Instead, it is their interaction that significantly impacts the dependent variable.

Interaction and Nonlinear Regression

Interactions can introduce non-linear models even with just one independent variable. Specifically, when an independent variable interacts with itself, it forms a squared interaction. This mathematical construct is akin to the square of the independent variable and, when visualized, results in a U-shaped or inverted U-shaped curve on a two-dimensional plane.

Consider the relationship between age and income: age influences the impact of age on income. For instance, income and age are positively associated for young and middle-aged individuals, but this association turns negative for individuals nearing retirement. This relationship can be roughly described as $Income = Intercept - Slope \times Age^2$.

When to Include Interaction

Deciding whether and which variables should interact with each other can be challenging. As the number of variables grows, the potential combinations of interaction terms proliferate rapidly. The number of independent variables engaged in an interaction can also vary. The earlier examples involved two-way interactions, but interactions can encompass any number of independent variables. However, multi-way interactions significantly escalate the model's complexity, a realm beyond this book's scope.

In the realm of academic research, adding interaction terms should not be driven solely by achieving a better fit. Excessive interaction terms can strain computational resources and yield results that are challenging to interpret. An excessive focus on model fit may result in overfitting, undermining the model's ability to generalize to new data. Therefore, the decision to incorporate interaction terms should be grounded in a robust theoretical framework or supported by existing evidence, akin to the justifications in examples such as “age and income” or “region and corruption.”

5.3 What's Next

In our next chapter, we will venture into the realm of classification tasks. We have laid the groundwork in this chapter by examining regression analysis, understanding how variables interact, and even exploring the nuances of interaction terms. This knowledge serves as a crucial foundation for delving into the complexities of classification. Logistic regression, one of our focal points, is an extension of linear regression—similar in methodology yet targeted at categorical outcomes. We will expand our understanding of regression models by exploring the intricacies of non-linear relationships and applying this concept specifically to classification scenarios. Additionally, the k-nearest neighbor algorithm, a staple in machine learning, will introduce a different

paradigm—focusing on the proximity of data points to classify new observations

5.4 Discussion Questions

1. How does the OLS method identify the best-fitting coefficients?
2. What are the key assumptions of linear regressions? Why are these assumptions crucial for model validity and reliability?
3. How do outliers affect regression analysis? What should we do about outliers?
4. What is the difference between multiple regression and multivariate regression?
5. What is multicollinearity, and why is it a concern in regression models?
6. Why and when should we add interaction terms in a regression model?
7. How does adding interaction terms impact the complexity and interpretability of regression models? Discuss the trade-offs in achieving a better fit versus a model's simplicity and ease of interpretation.
8. Can you think of real-world scenarios where understanding and implementing interaction terms in regression analysis could significantly impact decision-making or understanding relationships between variables?

In social science studies, estimating categorical or ordinal dependent variables is a common requirement. These variables often signify outcomes like the success or failure of an election campaign or individuals' sentiments measured on a Likert scale (ranging from “strongly disagree” to “strongly agree”). This task falls under the broader classification analysis, where the goal is to use an observation's independent variables to categorize the observation itself.

The robust linear regression models detailed in [Chapter 5](#) are invaluable but not designed to analyze non-continuous dependent variables. Consider [Figure 6.1](#), which portrays an OLS model demonstrating the influence of perceived corruption on a country's income group classification. Observations align along the horizontal axis but predominantly cluster within one of four levels on the vertical axis.

Despite the OLS model (improperly applied) suggesting a negative relationship between corruption perception and income groups, a challenge surfaces when estimating a country's income group. The linear model suggests a country with a corruption perception of 0.4 would land between upper middle income and high income. However, given the dependent variable's discrete four-category nature, the linear model inaccurately estimates the income group.

When estimating categorical or ordinal dependent variables, we should not treat categories as continuous values, which has proven inadequate. Instead, the approach estimates the probability of an observation belonging to a category with a linear model. This approach, known as generalized linear models (GLMs), falls under the parametric approach. Constructing a GLM for a discrete dependent variable entails assuming a specific probability distribution tailored to the dependent variable, followed by applying a link function to transform the relationship between independent and dependent variables from a nonlinear curve to a linear one.

The subsequent sections of this chapter delve into various GLMs commonly used in social science studies and nonparametric classification tools.

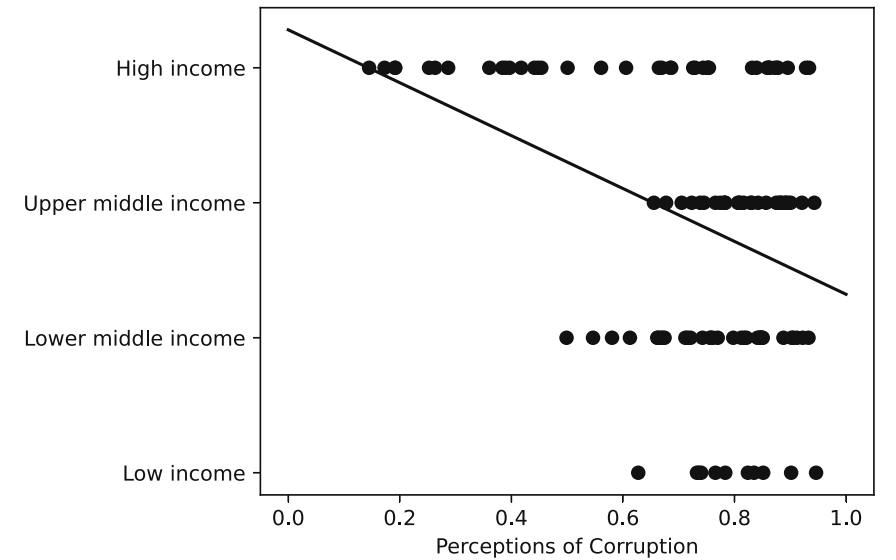


FIGURE 6.1
Forcing an OLS model on a categorical dependent variable.

6.1 Modeling Dependent Variables with Two Categories

Simple logistic regression, often called logit regression, comes into play when dealing with a dependent variable having two categories, typically a binary scenario like “yes” or “no.” Instances include determining if a person will vote, whether a visitor on an e-commerce website will click the “buy” button, or if a student will pass a course. Conventionally, the binary dependent variable takes on values of 1 and 0, where 1 represents the occurrence of an event, and 0 represents its absence. This regression model estimates the probability of the event occurring (value ranges from 0 to 1, or 100%). Independent variables in a logistic regression model can be categorical or continuous.

Mathematically, a simple logistic regression with one independent variable is expressed as:

$$\log \frac{P(Y = 1)}{1 - P(Y = 1)} = a + bx.$$

The left side of the equation is the log odds of the probability of the occurrence of the event, and the right side of the equation is a linear regression model, with a being the intercept and b the coefficient. Note that the output of the formula is log odds, the logarithm of odds. Odds are the likelihood of something occurring compared to that of it not occurring. As you may agree, changes

in log odds are intuitively more challenging to understand than changes in probability.

A simple mathematical transformation is applied to convert log odds into probability:

$$P(Y = 1) = \frac{1}{1 + e^{-(a+bx)}},$$

where e represents the base of the natural logarithm, while a stands for the intercept and b for the coefficient of the independent variable x . This formula might seem complex initially, but it consists of two components: $a + bx$ in the exponent of e , representing the linear model, and the remainder, a link function called the logistic or sigmoid function, transforming the linear model's result into a probability between 0 and 1.

It is crucial to note that determining coefficients a and b in logistic regression differs from the process in linear regression. The OLS method minimizes residuals to find a and b in linear regression. On the contrary, in logistic regression, the Maximum Likelihood Estimation (MLE) method identifies coefficients that maximize the alignment between estimated probability and observed data, effectively minimizing errors. This iterative process continues until it reaches an optimal state (converges). However, unlike the OLS method, convergence in logistic models is not guaranteed due to factors such as multicollinearity, outliers, and model complexity.

6.1.1 Example: Impact of Perceptions of Corruption on Binary Income Group

In this example, we will explore how perceptions of corruption (independent variable) affect a country's income group categorization (dependent variable). The challenge lies in the fact that income group is not binary. To proceed, we consolidate the "upper middle income" and "high income" categories into a new "upper income" category. Similarly, we merge the "lower middle income" and "low income" into a new "lower income" category. Transforming these categories into a dummy variable, we assign values of 1 to denote "upper income" and 0 for "lower income." Among the 114 countries in the 2021 dataset, 69 fall into the "upper income" category, while the rest are categorized as "lower income."

In [Figure 6.2](#), the plot demonstrates that countries categorized as "lower income" consistently exhibit corruption perceptions above 0.5. In contrast, countries in the "upper income" group display a broader distribution of corruption perceptions.

The logistic model can be mathematically described as:

$$\log \frac{P(\text{Being Upper Income})}{1 - P(\text{Being Upper Income})} = a + b \times \text{Perceptions of Corruption}.$$

After executing the regression analysis, the outcomes are presented in [Table 6.1](#). Interpreting the impact of the independent variable on the dependent

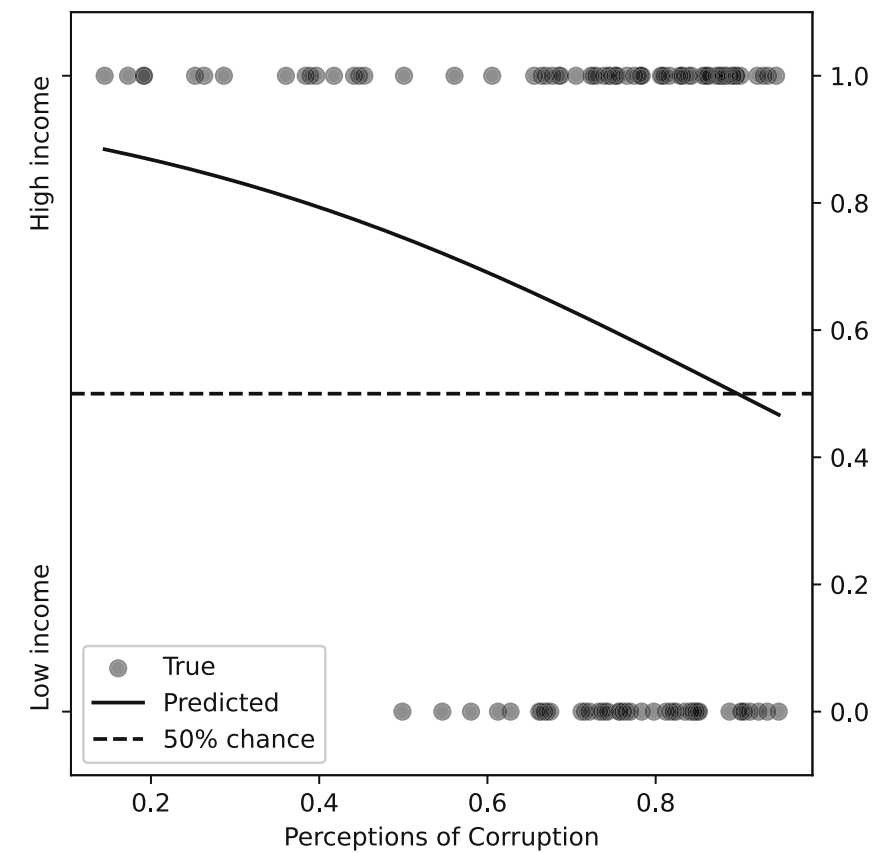


FIGURE 6.2
Impact of corruption perceptions on income groups.

variable, akin to the analysis of OLS regression results, directs our attention to the coefficient of the independent variable. Encouragingly, the coefficient for corruption perceptions exhibits statistical significance. However, as previously discussed, the output of the logistic regression model represents log odds. Consequently, the coefficient of the independent variable denotes that a unit increase in perceptions of corruption corresponds to a 2.7-unit decline in the log odds of being in the upper-income group.

Three simplified ways to interpret coefficients in log odds exist.

- First, suppose precision regarding the independent variable’s impact is less important than the direction of the impact. In that case, the negative sign of the log odds illustrates a negative relationship between corruption perceptions and income group.

TABLE 6.1
Relationship between income groups and corruption perception

	Coeff's in Log Odds	Coeff's in Odds Ratio
Corruption Perceptions	-2.7** (1.23)	0.067
constant	2.42** (0.95)	11.31
N	114	
LR Chi Squared p-value	0.017	

Standard errors in parentheses
* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

- Second, we can use software to convert log odds into an odds ratio. The odds ratio, representing the ratio between the odds of one event to another, aids in gauging the likelihood of an event relative to another event. To interpret the odds ratio in logistic regression practically, deducting one from the odds ratio and multiplying the difference by 100 reveals the percentage change in the odds of an event occurring for each unit increase in the independent variable.

In our example, the odds ratio for corruption perceptions in [Table 6.1](#) is 0.067. Calculating $(0.067 - 1) \times 100$ returns -93.3, indicating that a one-unit increase in perceptions of corruption leads to a 93% decline in the odds of a country being classified in the upper-income category. This decline does not signify a 93% reduction in the probability but instead denotes a decrease in the odds. Remember that probabilities range from 0 to 1, whereas odds extend from 0 to positive infinity.

- The third approach involves calculating and visualizing changes in the probability of the target event. Utilizing the logistic model to estimate the probability of a country being an upper-income country based on its corruption information, [Figure 6.2](#) reveals a nonlinear reduction in the probability of being in the upper-income category with increasing perceptions of corruption. Observations with lower perceptions of corruption predominantly align with upper-income countries, leading to a higher estimated probability. Conversely, observations exhibiting higher perceptions of corruption distribute more evenly between upper- and lower-income categories, resulting in an approximate 50% probability of being a high-income country.

6.1.2 Evaluating Goodness-of-Fit

The likelihood ratio test (LR test), or log-likelihood ratio test (LLR test), serves as a primary measure for evaluating the goodness-of-fit of a logistic regression model. This test compares the log-likelihood, which gauges how well a model's predictions align with observed outcomes, between our regression

model and a null model (a simpler model with only the dependent variable and an intercept). A larger log-likelihood indicates a better fit with observed data. The difference in log-likelihood between the two models signifies the improvement in model fit, which is assumed to follow a chi-squared distribution.

Subsequently, a chi-squared test is conducted to assess if our model exhibits significantly better fitness (measured in log-likelihood) than the null model. If the test's p-value is less than 0.05, it implies that our model outperforms the null model. In [Table 6.1](#), the LR test reports a p-value of 0.017, indicating superior fitness of our model compared to the null model.

Another metric often seen in software-generated logistic regression outputs is the pseudo-R-squared. While the traditional R-squared in linear regression quantifies the proportion of variation in the dependent variable explained by the model, the “pseudo” in pseudo R-squared highlights that it lacks a direct interpretation akin to the linear regression R-squared. Hence, although software generates this metric as part of the analysis, it is not commonly utilized to evaluate the goodness-of-fit in logistic regression.

6.1.3 Assumptions of Logistic Regression Analysis

As a generalized linear model, logistic regression has unique assumptions while sharing some assumptions with linear regression.

Assumption 1: Binary Dependent Variable. Logistic regression applies to models with a dependent variable holding only two values.

Assumption 2: Linearity. Linearity in logistic regression pertains to the linear relationship between the log odds of the dependent variable and the independent variables.

Assumption 3: Independent observations. Similar to linear regression, ensuring independence among observations mitigates bias in outcome estimation.

Assumption 4: Absence of Multicollinearity.

Independent variables should exhibit minimal correlation. Multicollinearity introduces bias in estimation and complicates result interpretation.

Assumption 5: No Complete Separation. Complete separation emerges when categories of the dependent variable that are linked to an independent variable or the combinations of independent variables exhibit no overlap. For instance, in [Figure 6.2](#), countries with high corruption perceptions display a mix of upper- and lower-income groups. If all upper-income countries in the dataset have low corruption perceptions while all lower-income countries exhibit high corruption perceptions, this scenario presents a complete separation issue. Complete separation presents a problem for logistic regression because it leads to the maximum likelihood estimates for the coefficients

becoming infinite (as a result of the model trying to align to perfect prediction).

6.2 Multinomial Logistic Regression

Simple logistic regression becomes inadequate when dealing with a dependent variable comprising three or more unordered categories. In such cases, the approach shifts to using multinomial logistic regression. It could be applied to estimate a voter's political party affiliation in a multi-party political system based on their stance on public issues. Streaming or social media platforms leverage this model to categorize user types or preferences derived from their platform activities or interactions, improving content recommendation and enhancing user experience.

Multinomial logistic regression operates on the same principles as logistic regression. It models the log odds of an observation belonging to each category with respect to a chosen reference category. Mathematically, it is expressed as:

$$\log \frac{P(\text{Category } k)}{P(\text{Reference Category})} = a_k + b_{1k} \times x_1 + b_{2k} \times x_2 + \dots + b_{nk} \times x_n, \quad (6.1)$$

where k represents one of the categories of the dependent variable, a_k stands for the intercept estimated for category k , x_1 denotes the first independent variable, b_{1k} signifies the estimated coefficient of x_1 for category k , and so forth.

To exemplify the functioning of multinomial logistic regression, let us utilize the region variable from the World Happiness Report as the dependent variable, given its seven regional categories. The independent variables will be the Ladder score and log GDP per capita. It is crucial to note that this analysis or model does not insinuate a causal relationship between ladder scores or log GDP per capita and a country's geographic location. Instead, it seeks to estimate the likelihood of a case belonging to a specific category based on the available data.

Prior to conducting the regression analysis, conversion of the region variable into numeric values is necessary. Assignment of numerical labels from 0 to 6 to each region can be done based on alphabetical order. The region assigned the lowest value, East Asia and Pacific, is treated as the reference region. It is important to emphasize that designating a different region as the reference point does not influence the statistical significance of coefficients.

The analysis yields a set of coefficients for each region, excluding the reference region. To examine the relationship between a specific region, say South Asia, and the independent variables, the mathematical formulation is

represented as:

$$\log \frac{P(\textit{South Asia})}{P(\textit{East Asia})} = a_{SA} + b_{1SA} \times \textit{Ladder Score} + b_{2SA} \times \log \textit{GDP per capita},$$

where the subscript “SA” refers to the coefficients for the South Asia region.

The results of the multinomial logistic regression are presented in [Table 6.2](#). Notably, the estimated coefficients are organized according to regions, portraying the difference between each region and the reference region. For instance, consider the Middle East and North Africa region. Its coefficient for the Ladder score is -1.6 and is statistically significant ($p < 0.05$). While holding all other variables constant, for every one-unit increase in the Ladder score, the log odds of a country being categorized as Middle East and North Africa, in contrast to East Asia, decreases by 1.6. Like logistic regression coefficients, these can be translated into odds ratios to enhance interpretability. A 1.6 decrease in log odds translates to an odds ratio of 0.2, or 80% decrease, calculated as $(0.2 - 1) \times 100$, in the odds of being categorized as Middle East and North Africa compared to East Asia. On the other hand, the coefficients for North Africa are deemed insignificant; alterations in the variables do not impact the log odds (or odds) of being categorized as North Africa as opposed to East Asia.

The most effective way to communicate the results to audiences is by visualizing each category’s probabilities rather than focusing solely on log odds or odds ratios. [Figure 6.3](#) on p. 110 demonstrates how the two independent variables influence the estimated probabilities for regions.

Given three variables (one dependent and two independent variables) and seven categories, an ideal representation would be in 3D, where each region’s probability is depicted by a surface within this environment. However, such a figure might be challenging to interpret in this book’s black-and-white print format.

To present the relationship in a 2D form, you can set one variable to a single or a limited number of fixed values. For instance, the three subplots of the Figure set the Ladder scores at the 25th, 50th, and 75th percentiles in the data. While these values are arbitrary, the objective is to showcase the impact of Ladder score changes on estimated probabilities. Only the probabilities of Latin America and Caribbean, Middle East and North Africa, and Sub-Saharan Africa are shown to simplify the figures and highlight distinct patterns.

As per the estimated results, when the ladder score is at the 25th percentile (top plot), Sub-Saharan Africa emerges as the most probable region at low GDP per capita levels. Conversely, the probability of being in the Middle East and North Africa region rises at high GDP per capita levels. At mid-range GDP per capita, the probabilities of these three regions are comparable.

In contrast, at the 50th percentile of ladder scores (middle plot), the probability of Latin American and Caribbean increases at mid-range GDP per capita. At the same time, the likelihood of Middle East and North Africa

TABLE 6.2
Relationship between regions and ladder score and log
GDP per capita

Europe & Central Asia	Log-odds	Odds Ratio
Ladder Score	0.054 (0.66)	1.05
Log GDP per capita	1.3* (0.75)	3.67
constant	-12.1** (5.04)	0
Latin American & Caribbean		
Ladder Score	1.28* (0.75)	3.58
Log GDP per capita	-1.05 (0.73)	0.35
constant	2.84 (4.67)	17.08
Middle East & North Africa		
Ladder Score	-1.6** (0.71)	0.2
Log GDP per capita	1.55* (0.89)	4.74
constant	-6.4 (6.53)	0.001
North America		
Ladder Score	0.23 (1.8)	1.26
Log GDP per capita	3.83 (2.37)	46.05
constant	-43.27* (22.41)	0
South Asia		
Ladder Score	-1.89** (0.85)	0.15
Log GDP per capita	-0.54 (1.05)	0.58
constant	13.16 (8.57)	518276
Sub-Saharan Africa		
Ladder Score	-1.08 (0.72)	0.34
Log GDP per capita	-2.05** (0.81)	0.13
constant	24.16*** 6.33	3.1×10^{10}
N	111	
LLR Chi-Squared p-value	0	

Standard errors in parentheses
* p < 0.1, ** p < 0.05, *** p < 0.01

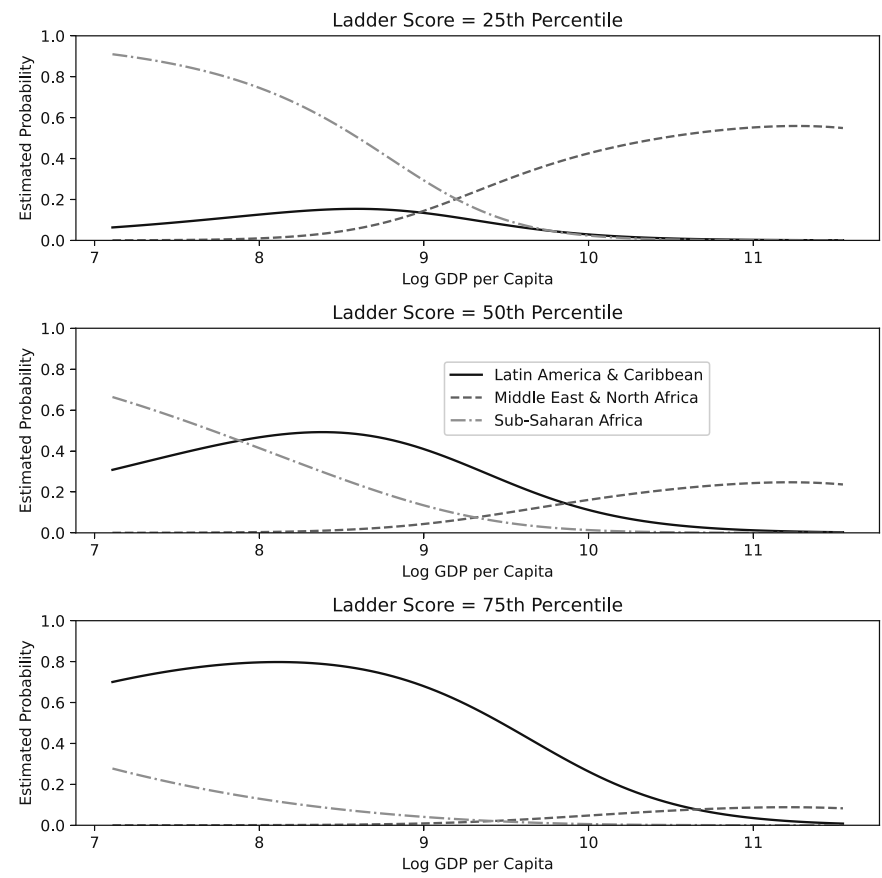


FIGURE 6.3
Estimated probabilities for three regions based on ladder scores and log GDP per capita.

diminishes at higher GDP per capita levels. At a higher ladder score (75th percentile, bottom plot), the Latin American and Caribbean region becomes notably more likely at low- to mid-range GDP per capita levels. At high GDP per capita levels, none of the three regions exhibit a high likelihood.

It is important to reiterate that the results of this example do not imply a causal relationship between ladder scores or log GDP per capita and a country's geographic location. Rather, they highlight the probability of a case belonging to a category based on the data available.

6.3 Ordered Logistic Regression

Ordered logistic regressions are suitable for examining ordinal dependent variables with more than two levels. This method is commonly employed in analyzing people's preferences, like the extent of their agreement or disagreement with a statement (measured using Likert scales such as "strongly disagree," "disagree," "neither agree nor disagree," "agree," and "strongly agree"), or their affinity towards a product or service (measured by the number of stars). Due to the hierarchical nature of the dependent variable, it is imperative to ensure a clear hierarchical order within the categories. Failure to do so might lead to assigning category order based on alphabetical order instead of the intended qualitative order of the categories.

The mathematical expression for ordered logistic regression is:

$$\log \frac{P(\text{Category} \leq k)}{P(\text{Category} > k)} = a + b_1 \times x_1 + b_2 \times x_2 + \dots + b_n \times x_n,$$

where k represents one of the categories within the dependent variable, a stands for the intercept, and b 's denote the coefficients of the independent variables. The left side of the equation defines the log odds of being in a specific category or lower (also called cumulative odds) in contrast to the above categories. In comparison, the left side of the multinomial regression in Eq. 6.1 on p. 107 estimates the log odds of being a specific category as opposed to the rest.

Examining the impact of life expectancy, freedom, and generosity on income groups involves a dependent variable comprising four ordinal categories: low income, lower middle income, upper middle income, and high income, making it suitable for ordered logistic regression analysis.

Table 6.3 presents the regression outcomes, encompassing three distinct sections. The first section outlines the independent variables' coefficients (or odds ratios) alongside their statistical significance. Only the life expectancy coefficient (0.55) demonstrates statistical significance among these variables. This value signifies that each one-unit rise in life expectancy is associated with a positive increase (since the coefficient is positive) of 0.55 units in the log odds of being in a higher income group category relative to a lower category while holding other independent variables constant. The odds ratio of 1.73 for life expectancy indicates that every one-unit increase in life expectancy is linked with a 73% increase, computed as $(1.73 - 1) \times 100$, in the odds of being in a higher income group category.

Diving into the middle section, it focuses on the cutpoints or thresholds. In our ordered logistic regression model, the model leverages the four income categories to estimate a latent (unobserved) continuous variable, given the nature of the model as a generalized linear model. The three cutpoints demarcate the separation between the four income group categories. As per the

TABLE 6.3
Relationship between income group and life expectancy, freedom, and generosity

	Coefficients	Odds Ratios
Life Expectancy	0.55*** (0.08)	1.73
Freedom	3.24 (2.11)	25.45
Generosity	-2.36 (1.47)	0.094
Cutpoints / Thresholds		
Low / Lower Middle Income	33.19	
Lower Middle / Upper Middle Income	38	
Upper Middle / High Income	40.31	
N	110	
LLR Chi-squared p-value	0	

Standard errors in parentheses
* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

results, the cutpoint between the low-income and lower-middle-income categories is 33.19, while the cutpoints for the higher two categories are 38 and 40.31, respectively. However, interpreting these values is not imperative, as the exact nature or significance of the latent variable remains unknown. The cutpoints are outcomes generated by the model based on the available data and can be deemed a byproduct of the analysis.

The bottom section offers insight into the log-likelihood test’s sample size and the p-value. A p-value of 0 signifies a substantial enhancement in model fit compared to a null model.

6.3.1 The Parallel Regression Assumption

One striking dissimilarity between ordered and multinomial logistic regression is their coefficient presentation. The former entails only one set of coefficients, implying that the impact of an independent variable across various levels of the dependent variable remains consistent. Conversely, multinomial regression produces distinct sets of coefficients for each category of the dependent variable, suggesting varied impacts of independent variables across different categories.

This behavior in ordered logistic regression is due to a pivotal assumption: the parallel regression assumption. This premise asserts that the effects of independent variables on the odds of the outcome remain uniform across different levels of the dependent variable.

Validating the parallel regression assumption is crucial when we do not know the population. Unfortunately, as of now, dedicated Python

functions or libraries for this test are absent. Hence, a manual examination is required.

The examination entails performing an ordered logistic regression and subsequently comparing its log-likelihood (model fitness) with that of a multinomial logistic regression with the same variables. The reasoning is rooted in the nested relationship between these models: the ordered logistic model imposes constraints on consistent proportional odds across the dependent variable levels, rendering it a subset of the more relaxed and precise multinomial model. In other words, if the constraints are removed, ordered logistic regression becomes multinomial logistic regression.

You may wonder if multinomial logistic regression is more relaxed and precise than an ordered logistic model, why do we even bother using the latter? The ordered logistic model may be preferred under the principle of parsimony, favoring simpler explanations that demonstrate comparable model goodness-of-fit. Thus, if the ordered logistic model does not violate the parallel regression assumption and showcases a similar fit to the more complex multinomial model, it becomes the preferred choice, trading precision for simplicity.

The comparison of model goodness-of-fit between the ordered and multinomial models is executed through the likelihood ratio test (discussed in [Section 6.1.2](#) on p. 105). The null hypothesis of the test is that the two models have no difference in goodness-of-fit. Suppose the difference in goodness-of-fit between the two models is statistically significant ($p\text{-value} < 0.05$). In that case, the ordered logistic model may violate the parallel regression assumption because the multinomial model has a better model fit. However, a non-significant difference ($p\text{-value} > 0.05$) suggests adherence to the assumption, and the two models have similar goodness-of-fit.

In evaluating our ordered logistic regression model from [Table 6.3](#), the chi-squared test yields a $p\text{-value}$ of 0.1. Given this result, the ordered logistic model appears to abide by the parallel regression assumption. Visually, [Figure 6.4](#) reinforces this conclusion, depicting strikingly similar estimated probabilities from both the ordered and multinomial regression models as life expectancy changes (life expectancy is chosen for the x-axis because it is statistically significant in both models). This alignment underscores the model's adherence to the parallel regression assumption. If the ordered logistic regression model violates the assumption, its estimated probabilities would differ significantly from those of the multinomial logistic regression model.

The technical details of ordered logistic regression are explained in [Section 14.2.3](#) on p. 275.

6.4 Non-Parametric Approach: K-Nearest Neighbors Algorithm

The k-nearest neighbors (KNN) algorithm is a powerful yet straightforward machine-learning technique for classification. Unlike parametric methods,

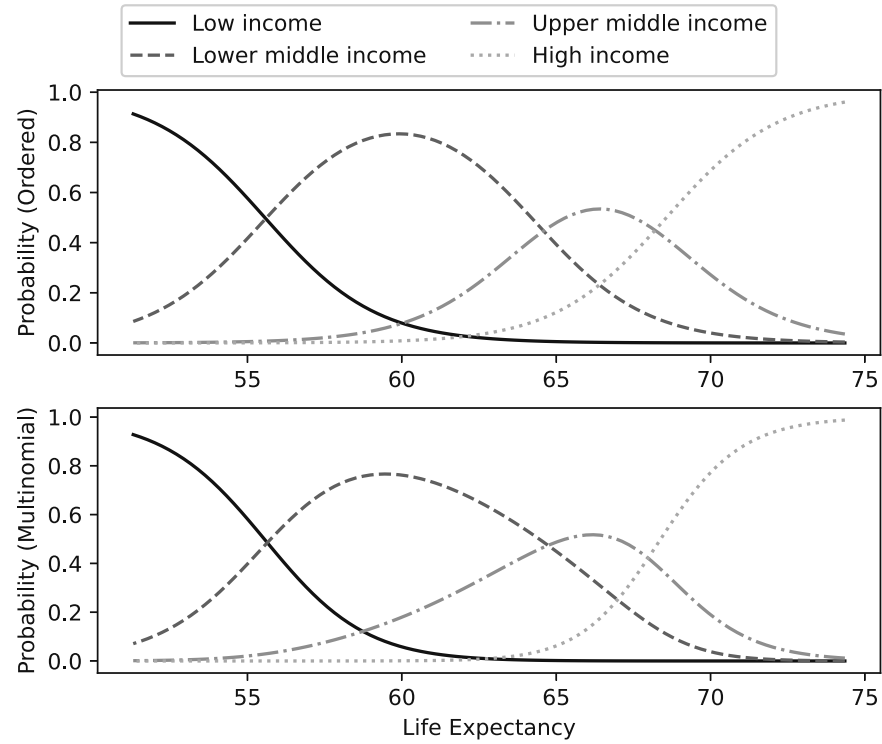


FIGURE 6.4
Estimated probabilities under the ordered logistic regression (upper) and multinomial logistic regression (lower).

KNN does not assume any specific data distribution or construct a model with calculated coefficients. Instead, it stores the data and computes results during estimation.

Let us look at a simple example to grasp how the algorithm operates. Figure 6.5 shows 11 data points consisting of five squares, five triangles, and one solid circle representing unknown data.

Based on the scatterplot, what would be your guess for the unknown data: a square or a triangle? If you chose a triangle, it is likely because the unknown observation is closer to the triangles than the squares. KNN follows a similar principle. The algorithm identifies the observations most similar to the unknown data and utilizes their labels to estimate that of the unknown data point. The parameter k in KNN determines the number of similar observations considered for estimation.

A distance metric measures the similarity between observations and unknown data. The shorter the distance between two data points, the more

similar they are; conversely, longer distances indicate less similarity. In this context, the similarity is measured with the Euclidean distance.

The Euclidean distance calculates the straight-line distance between two points, a concept you may recall from high school geometry. The general formula for the Euclidean distance d_E between two data observations with more than two variables can be expressed as:

$$d_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + \dots + (z_1 - z_2)^2},$$

where x , y , and z represent variables, and the subscript numbers denote the first and second observations. For instance, x_1 , y_1 , and z_1 represent the values of the first observation for the three variables, respectively, and x_2 , y_2 , and z_2 are the values for the second observation.

6.4.1 The KNN Algorithm

When we apply the KNN algorithm to determine the shape of the unknown data point in [Figure 6.5](#), the algorithm operates as follows: it initially computes the Euclidean distances between the unknown data point and each observation in the dataset. Subsequently, these distances are sorted in ascending order. If, for instance, we opt to utilize the three most similar observations for estimation, the algorithm selects the three observations closest to the unknown data

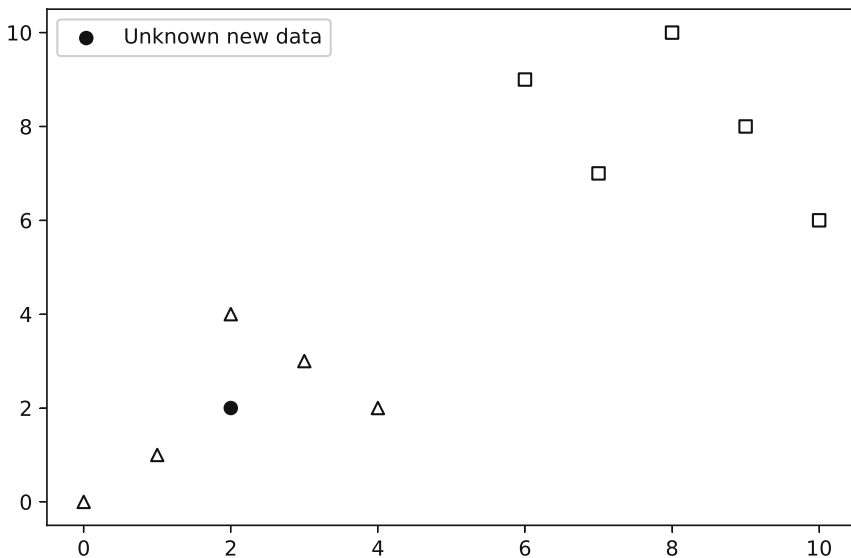


FIGURE 6.5

Estimate the shape of the unknown new data.

point, records their labels, and, similar to a voting mechanism, designates the label that appears most frequently among these three observations as the estimated label for the unknown data point. In our example, all three nearest observations to the unknown data point are triangles, concluding that our estimated shape is a triangle.

Various considerations influence the selection of the parameter k . First, when adopting the majority voting method, choosing an odd k value ensures the existence of a majority label value. In contrast, an even k could result in an equal count of different label values among the nearest neighbors, making it challenging for the algorithm to reach a decision. Second, the performance of the KNN can be adjusted by fine-tuning the k value to achieve optimal estimation results, a topic we will explore further in this chapter.

6.4.2 Processes of Performing KNN Classification

- The initial step in performing many machine-learning algorithms, including KNN, involves randomly partitioning the dataset into four sub-datasets: training and testing datasets for the independent variables and for the dependent variable. The rationale behind creating training and testing datasets is to simulate a real-world scenario where we have observed data with known label values and new data with unknown label values. The training dataset serves as the observed data, while the testing dataset simulates new data. We assume that we are unaware of their label values and employ KNN to estimate them. Given that we have the actual label values for the testing dataset, we can compare the estimated labels with the actual ones to evaluate the performance of our estimation. Based on this performance evaluation, we may adjust the k value.
- The second step involves normalizing the values of the independent variables. Normalization is a scaling process that transforms data to have a mean of 0 and a standard deviation of 1. Normalized data facilitate easy comparison and analysis, even if the original values differ in scales or units of measurement.
- In the third step, we apply the KNN algorithm to the training datasets to train the KNN classifier. The classifier “learns” the relationship between distances and label values. Subsequently, the trained KNN classifier is employed on the independent variables of the testing dataset to generate estimated label values for the dependent variable. These estimated label values are then compared with the actual label values in the testing dataset, typically using a confusion matrix, to evaluate the model’s performance.

This step completes one iteration of training a KNN model. However, since the choice of the k value significantly influences the performance of a KNN model, it is essential to test KNN models with different k values and identify

TABLE 6.4
Confusion matrix for label values in two categories

Actual		Estimated	
		Positive	Negative
	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

the k that yields the best estimation performance. Ultimately, the “best” k is employed to train the final KNN model for estimation.

Confusion Matrix

A confusion matrix serves as a contingency table that provides a comprehensive summary of the performance of a machine-learning classification model on a test dataset. The size of the table is $n \times n$, where n represents the number of categories in the dependent variable. Consider Table 6.4, which illustrates a 2×2 confusion matrix for binary classification.

The columns and rows of the table represent estimated and actual label values. The terms “positive” and “negative” are conventional generic labels for binary categories and can be substituted with the actual category names, such as “Category A” and “Not Category A.” The four distinct situations resulting from the model’s estimations are as follows:

True Positive (TP): The count of cases in the testing dataset that the KNN algorithm estimates as the “Positive” category, and their actual label values are also “Positive.”

True Negative (TN): The count of cases where the algorithm estimates the category as “Negative,” and their actual label values are also “Negative.”

False Negative (FN): The count for cases where the algorithm estimates a case to be “Negative,” but its actual label value is “Positive.”

False Positive (FP): The count for cases where a case is estimated to be “Positive,” but its actual label value is “Negative.”

The confusion matrix becomes more intricate for a dependent variable with more than two categories. Table 6.5 presents an example of a confusion matrix for three categories. In a multi-category confusion matrix, attention is given to the estimation performance for each category.

For instance, when an observation’s actual category is A and is classified by the algorithm as category A, it is considered a “True Positive” for category A (denoted as TP_A). If the same observation is incorrectly classified as category B, it is denoted as $Error_{AB}$, indicating that the actual category is A, but the estimated category is B.

The False Negative (FN) count for a category equals the sum of counts in the **row** of the category, excluding the True Positive (TP) count. For Category

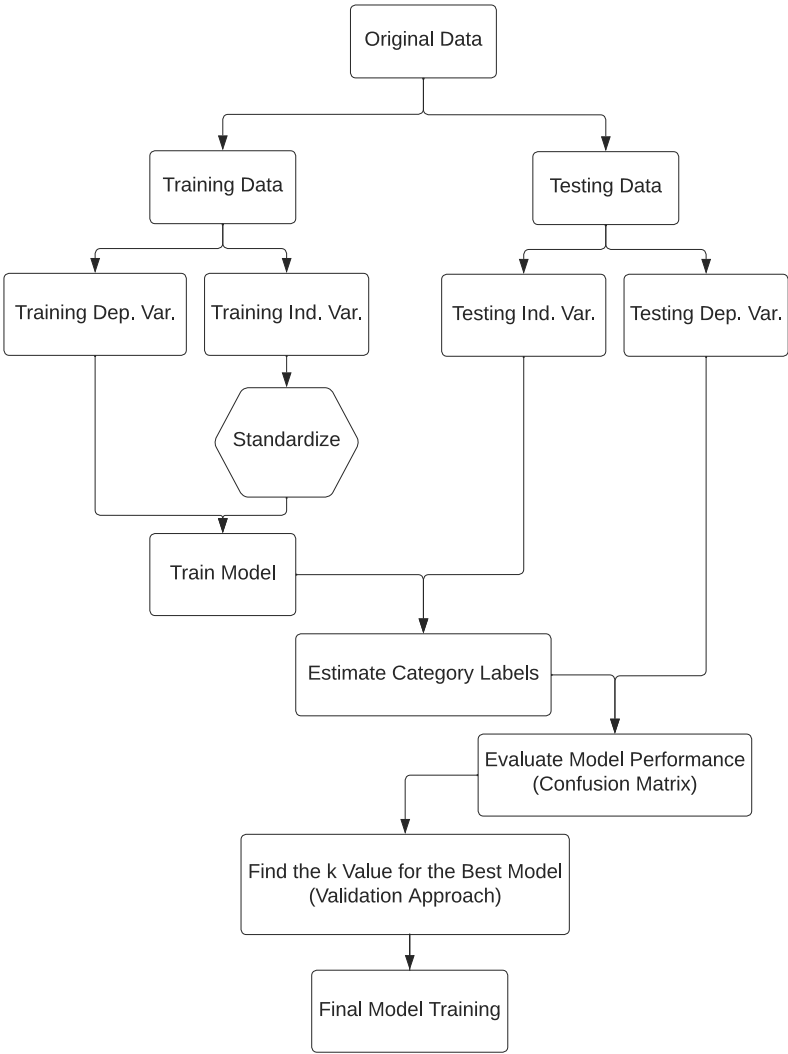


FIGURE 6.6
Process of KNN classification.

TABLE 6.5
Confusion matrix for label values in multiple categories

Actual	Estimated			
	Category A	Category B	Category C	
	Category A	Category B	Category C	
	Category B	Category C		
	TP _A	Error _{AB}	Error _{AC}	
	Error _{BA}	TP _B	Error _{BC}	
	Error _{CA}	Error _{CB}	TP _C	

A, $FN_A = Error_{AB} + Error_{AC}$, as the estimated categories are not Category A, and the estimations are false.

The False Positive (FP) count for a category equals the sum of counts in the category **column**, excluding the TP count. Thus, the False Positive count for Category A is $FP_A = Error_{BA} + Error_{CA}$, as the estimated categories are A (Positive), but the estimations are false.

The True Negative (TN) count for a category equals the sum of counts in the table, excluding both the row and column of the category. For Category A, its True Negative count is calculated as $TN_A = TP_B + Error_{BC} + Error_{CB} + TPC$.

Evaluating Confusion Matrix

These four values play a crucial role in calculating metrics that evaluate the performance of classification estimations. Let us delve into some of these metrics.

Accuracy: This metric gauges the overall correctness of the estimations. It is calculated as the ratio of correctly classified observations from the testing dataset to the total number of observations in the testing dataset. The mathematical expression for a binary classification scenario is:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN},$$

if the dependent variable has two categories.

For a scenario with more than two categories (as in [Table 6.5](#)), accuracy is calculated as:

$$\text{Accuracy} = \frac{TP_A + TP_B + \dots + TP_Z}{\text{Sum of All Counts in the Table}}.$$

Accuracy provides a comprehensive measure of the model’s correctness across all categories and can be a helpful metric when we have roughly equal numbers of “Positive” and “Negative” observations. When the frequencies of the categories are imbalanced, such as having the majority of observations actually “Positive,” a high accuracy value may not be reliable because it can result from simply guessing “Positive” for all observations. In such cases, precision and recall are better measures.

Precision: Precision measures the proportion of total correct “Positive” classifications. It is calculated as the ratio of true positive observations to the sum of true positives and false positives. Mathematically, for binary classification:

$$\text{Precision} = \frac{TP}{TP + FP}.$$

When there are more categories in the dependent variable, we calculate and analyze the precision metric for each category rather than the entire model.

$$\text{Precision}_A = \frac{TP_A}{TP_A + FP_A}$$

We focus on precision when the cost of false positives is high.

Recall (sensitivity or true positive rate): Recall measures the ability of the model to capture all the positive instances. It is calculated as the ratio of true positive observations to the sum of true positives and false negatives. For binary classification:

$$\text{Recall} = \frac{TP}{TP + FN}.$$

For the dependent variable with multiple categories:

$$\text{Recall}_A = \frac{TP_A}{TP_A + FN_A}.$$

We look at recall when the cost of false negatives is high.

F1 score: The F1 score is the harmonic mean of precision and recall. It provides a balance between precision and recall, especially in scenarios where there is an imbalance between positive and negative classes. The formula for binary classification is:

$$\text{F1 score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

For multiple categories:

$$\text{F1 score}_A = \frac{2 \times \text{Precision}_A \times \text{Recall}_A}{\text{Precision}_A + \text{Recall}_A}$$

Fall-out (or false positive rate): Fall-out measures the proportion of observations incorrectly classified as “Positive” among the total number of actual “Negatives.” Mathematically:

$$\text{Fall-out} = \frac{FP}{FP + TN}.$$

For multiple categories,

$$\text{Fall-out}_A = \frac{FP_A}{FP_A + TN_A}.$$

Specificity (also called true negative rate): Specificity is the complement fall-out. It measures the proportion of observations correctly classified as “Negative” among all the observations that are actually “Negative.” Mathematically,

$$\text{Specificity} = \frac{TN}{TN + FP}.$$

It is important to note that specificity is used for binary dependent variables and is unsuitable for those with more than two categories.

How to Select the k Value

The selection of the k value plays a crucial role in determining the performance of the KNN model. However, the relationship between the k value and model fit is not linear. You can leverage domain expertise to identify the optimal k for achieving the best model fit. In cases where domain expertise is lacking, a technical solution involves validation. Through validation, multiple KNN models with different k values are tested, and their performances are compared to determine the k that yields the best results.

One common validation technique is hold-out validation (or train-test split validation). In this approach, the original data set is repeatedly split into training and testing sets, with each iteration utilizing a different k value. This process generates various KNN models and their corresponding confusion matrices. Performance metrics, such as accuracy, are derived from these matrices, and the k value associated with the best performance is selected for the final model.

While hold-out validation is simple, straightforward, and efficient, it has drawbacks. Model performance can be sensitive to the random data split, especially with smaller datasets. There is also a risk of overfitting, as the focus is solely on identifying the best model fit.

A more advanced validation technique is cross-validation. In cross-validation, the data set is divided into multiple sub-datasets or folds. The KNN analysis is performed multiple times, with each fold serving as the testing dataset and the remaining folds as the training dataset in each iteration. This process is repeated with different k values, and the optimal k is determined based on overall performance.

If sensitivity to random data splits persists, cross-validation can be repeated multiple times, and the most frequently occurring optimal k value across repetitions can be chosen as the final k value. Cross-validation provides a more robust estimate of model performance, addressing issues related to sensitivity and overfitting. However, it comes at the cost of increased computational requirements, particularly for larger datasets.

The KNN algorithm offers advantages such as ease of implementation, interpretability, and handling of nonlinear relationships. However, it is not without drawbacks. Sensitivity to noise and outliers, high memory and

TABLE 6.6
KNN classification report

	Precision	Recall	F-1	Support
High Income	0.44	0.67	0.53	6
Upper Middle Income	0.56	0.45	0.5	11
Lower Middle Income	0.75	0.3	0.43	10
Low Income	0.17	1	0.29	1
Accuracy			0.46	28
Macro Average	0.48	0.61	0.44	28
Weighted Average	0.59	0.46	0.47	28

computation requirements, and careful selection of the k value and the distance metric should be considered when employing the KNN algorithm.

6.4.3 Example: Estimating Income Group Classification

In this example, we employ the KNN algorithm to estimate the income group classifications of countries based on their life expectancy and freedom scores.

Start by normalizing the original dataset to ensure uniformity. Subsequently, a quarter of the data is randomly designated as the testing dataset, with the remainder assigned as the training dataset. The optimal k value for the KNN model is determined through a cross-validation process using 5 folds, ranging the k values from 1 to 9. Model performance is evaluated using the accuracy metric.

The cross-validation results reveal that a k value of 8 yields the best overall model performance. This optimal k value is then applied in the final KNN model to generate estimations. Table 6.6 presents various metrics reflecting the model’s performance.

Given the four categories of income groups, precision, recall, and other metrics are reported individually for each category. For instance, the model exhibits a precision of 0.44 for the high-income category, signifying that 44% of the cases classified as high-income by the model are indeed high-income. The recall for the high-income category is 0.67, indicating that the model accurately identifies 67% of instances belonging to this category.

An overall accuracy metric is provided, representing the percentage of correctly estimated categories in the test dataset. In this instance, the KNN model achieves an accuracy of 0.46, correctly estimating 46% of the categories.

The macro-average values, depicted in the respective row, represent the arithmetic mean of metrics across all categories and measure overall model performance. For example, the macro-average precision is 0.48, implying that, on average, 48% of the model’s estimated classifications are correct.

The “support” column in Table 6.6 provides insights into the sizes of each category. For instance, the original dataset comprises 110 observations, with

a quarter (28) allocated for the testing dataset. Within these 28 observations, the distribution across income categories is as follows: 6 for high-income countries, 11 for upper-middle-income countries, 10 for lower-middle-income countries, and only 1 for low-income countries.

It is crucial to note that the macro-average treats precision values for all categories with equal weight, assuming similar category sizes in the test dataset. When category sizes are imbalanced, as is the case here, the precision of categories with a larger number of observations is more influential in determining the overall model precision than those with fewer observations. Therefore, a weighted average precision is preferred to obtain a more representative measure of the overall model precision in such imbalanced scenarios. This approach appropriately considers the impact of each category's precision in proportion to its respective size, providing a more nuanced evaluation of the model's performance.

6.4.4 The Receiver Operating Characteristic (ROC) Curve and Area Under ROC Curve (AUC)

The Receiver Operating Characteristic curve (ROC curve) provides a visual means of assessing the performance of binary classification models, such as logistic regressions and binary KNN models. These can be converted into multiple binary classification models for multi-category classification models. It is essential to note that an ROC curve is primarily used for evaluating classification model performance and is not intended for hypothesis testing.

The ROC curve illustrates the recall (true positive rate) against the fall-out (false positive rate) at different decision thresholds. Decision thresholds determine how a model assigns category labels to data points. For instance, in a binary KNN model, the category label is assigned based on the majority rule, where the label that constitutes more than 50% of the neighbors' labels is selected. Similarly, the category with more than a 50% chance is chosen as the final estimation in logistic regression.

The ROC analysis involves adjusting the decision threshold across various levels (from 0 to 1) and observing how model performance changes at different thresholds. ROC curves, depicted in [Figure 6.7](#), enable the comparison of model performances. A diagonal line represents the performance of a model that makes random guesses. A curve positioned higher above the dashed diagonal line indicates superior estimation performance. With perfect performance, the ideal model reaches the top-left corner (0,1).

In the real-data context of [Section 6.1.1](#), where income groups are merged into two categories, [Figure 6.8](#) compares the ROC curves of logistic and KNN models. Both curves surpass the dashed diagonal line, signifying better performance than random guessing. However, the overlap makes evaluation challenging. Here, we need to focus on the area under the ROC curve (AUC) metric, ranging from 0 to 1.

The AUC quantifies the area under the ROC curve, where 0 suggests entirely incorrect estimations, 0.5 (dashed diagonal line) equates to random

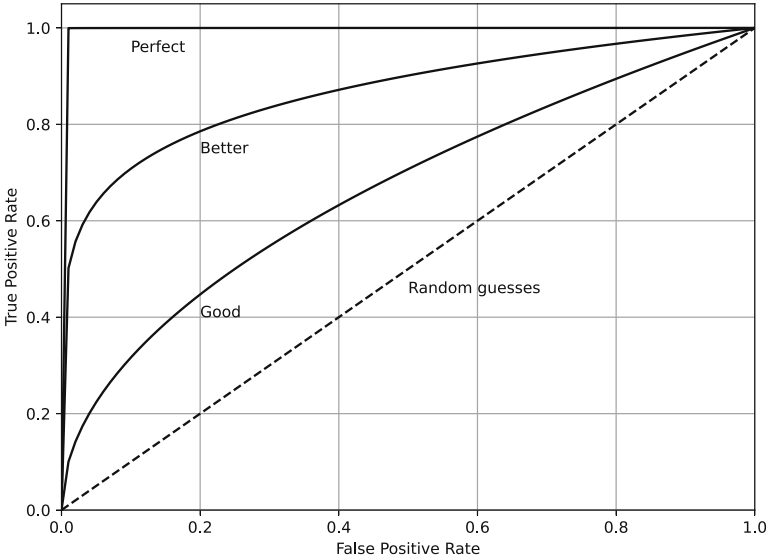


FIGURE 6.7
ROC curves and estimation performances.

guessing, and 1 signifies perfect estimations. A higher AUC generally indicates better classification. In this example, the logistic model boasts an AUC of 0.74, while the KNN model achieves an AUC of 0.79. Therefore, both the logistic and KNN models can be employed in the estimation of binary income groups, but the latter exhibits superior classification performance.

6.5 Stereotyping in Classification Models

When employing a classification model to categorize cases, it is crucial to remain vigilant about the issue of stereotyping. Stereotyping occurs when a model unfairly generalizes characteristics or behaviors of specific groups, leading to biased and potentially discriminatory outcomes. A recent incident highlights this concern, where an Asian student used an AI platform to generate a professional headshot, only to receive an image with light skin and blue eyes, features typically associated with individuals of Caucasian descent [Bha23].

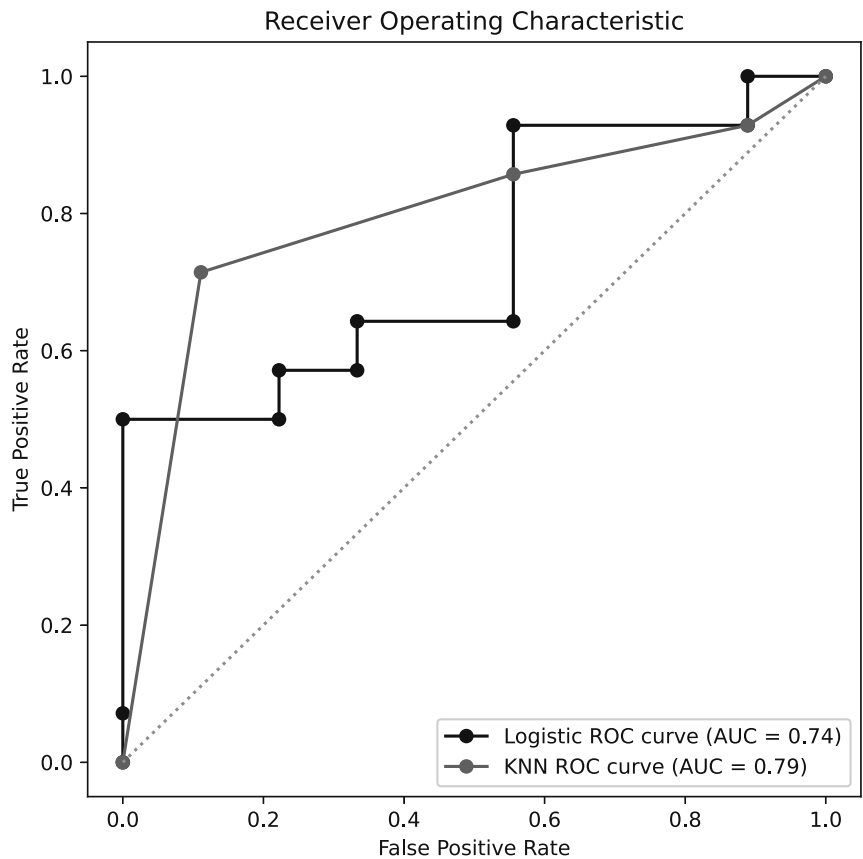


FIGURE 6.8
ROC curves of logistic and KNN models.

One of the root causes of stereotyping lies in biases present in the training data. Ensuring that the training data is representative and diverse is essential. For example, data used in social science studies may unintentionally incorporate pre-existing social biases or discriminatory patterns. If a model defining “professionalism” is trained using media photos, then there is a risk of a disproportionate representation of individuals of Caucasian ethnicity. Similarly, if the model relies on surveys to assess “professionalism” but lacks diversity in the respondent pool, it can introduce bias.

Variable selection is another source of stereotyping. Thoughtful consideration of the variables used to measure a phenomenon is essential. Avoiding variables likely to introduce biases or reinforce stereotypes is crucial in developing fair models.

After model development, it is vital not to focus solely on fitness but also to perform comprehensive assessments of the model's estimations across various demographic groups. This approach ensures fairness and helps mitigate the impact of stereotyping. Awareness and proactive measures at each stage of the modeling process are key to addressing and preventing stereotyping in classification models.

6.6 What's Next

The next chapter introduces complex network analysis, focusing on the construction, visualization, and interpretation of complex networks from relational data, using mathematical graphs to represent entities and their connections. It covers key concepts including network types, centralities, community detection, and bipartite networks, providing foundational knowledge for understanding and analyzing the structure and dynamics of complex networks in various contexts.

6.7 Discussion Questions

1. Simple logistic regression, multinomial logistic regression, and ordered logistic regression are tailored to tackle distinct problems. How do we determine the most suitable method for a specific problem?
2. How does the interpretation of coefficients in logistic regression differ from that in linear regression?
3. What is the parallel regression assumption? Why is it important to test it when we perform an ordered logistic regression?
4. Discuss the significance of choosing the k value in the K Nearest Neighbors (KNN) algorithm. How does the choice of k influence the model's performance, and what strategies can be employed to determine the optimal k value?
5. Compare and contrast hold-out validation and cross-validation as methods for assessing the performance of classification models. What are the strengths and weaknesses of each approach, and in what scenarios would one be preferred over the other?
6. Could you provide an example of stereotyping? What measures can be taken to prevent stereotyping in that scenario?

Complex Network Analysis

Tabular data, where each row represents a sample and each column a specific property, is highly effective for characterizing individual entities like people, organizations, firms, and nations. However, its utility diminishes when describing relational data involving multiple entities, such as human friendships, international trade relations, and corporate partnerships. In these scenarios, complex networks, particularly social networks, offer a more suitable framework for organizing relational data.

7.1 General Definitions

In network analysis, a network is essentially a mathematical graph or its visual representation. The latter is often called a sociogram.

A mathematical graph, as conceptualized in graph theory, comprises a set of discrete nodes, which represent individual entities, and a set of edges, denoting the relationships between these entities.

Depending on the specific field or context, nodes and edges may be referred to by various terms. For instance, nodes are known as vertices in graph theory, points in spatial networks, actors in social networks, units or elements in abstract networks, and stations or sites in communication networks. Similarly, edges are often called links in computer, communication, and social networks; connections in interaction networks; ties in social networks; and arcs in directed networks. Other terms include bonds, pathways, and bridges.

Typically, nodes are depicted as circles (either hollow or filled), while edges are depicted as line segments or arcs. We will primarily use the terms “node” and “edge,” unless there is a need to highlight certain aspects.

Figure 7.1 on p. 128 shows a sociogram of the classical “karate” dataset: the recorded social ties among the members of a university karate club. The dataset was collected by Wayne Zachary in 1977 [Zac77].

A network often includes additional attributes for nodes and edges, providing more detailed characterization. You can employ visual elements like size, color, and thickness to convey various attributes. Line thickness and style might indicate the strength and type of relationships, with arrows showing directionality. Node shape and size could represent factors such as n

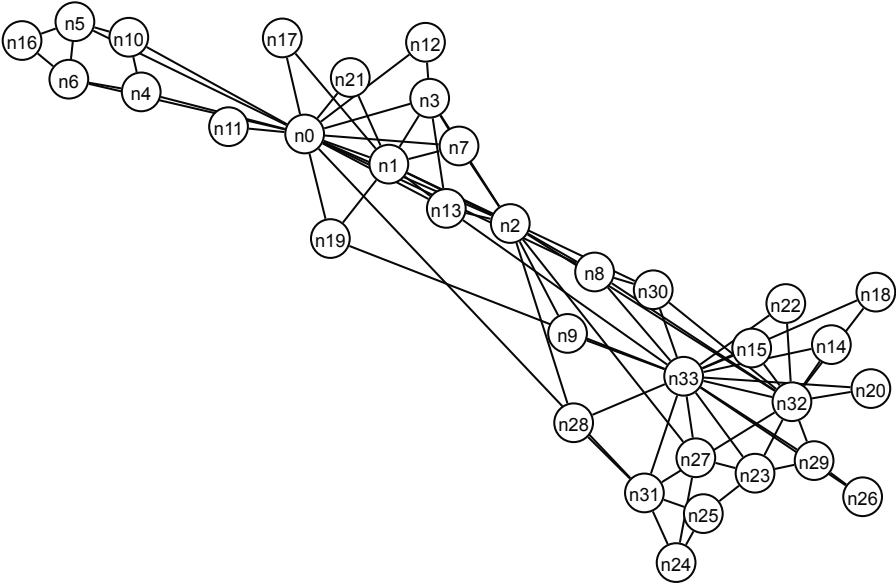


FIGURE 7.1
Zachary karate club network.

individual's gender and age, while color might signify their social roles. Thus, a network can be viewed as a mathematical graph enriched with domain-specific attributes for nodes and edges.

7.1.1 Regular and Complex Networks

A complex network is defined by its distinctive pattern of nodes and interconnections, which typically lacks regularity. These networks are often complex in structure and not easily described by straightforward formulas. In contrast, regular networks, like grids and rings, exhibit predictable, uniform patterns. These simpler networks can generally be studied using traditional analytical methods, often without extensive computational or statistical approaches. Regular networks are less common in social contexts, underscoring the relevance of focusing on complex networks, especially in areas where irregular and intricate connections are more representative of the underlying structure.

Social networks, widely recognized within social science, are a specific category of complex networks. Since a social network is usually inherently a complex network, though not all complex networks are social, we will concentrate on complex networks in this chapter, encompassing their definitions, uses, characteristics, and Complex Network Analysis (CNA), also known as Social Network Analysis (SNA) in the context of social networks.

CNA aims to uncover and analyze the patterns of binary relationships between nodes. This understanding often reveals critical players, groups, or structures within the network.

By mapping out complex networks, CNA helps to identify which nodes are most central or influential within the network in terms of the flow of information, or act as bridges between different groups.

SNA is used to study how information, ideas, behaviors, and resources circulate through networks. This knowledge can be vital for understanding phenomena like the spread of news, the adoption of innovations, or the transmission of diseases. SNA can reveal the presence of subgroups or communities within larger networks, showing how they interact, how cohesive they are, and what roles different members play.

Lastly, by analyzing a network at different time points, SNA can provide insights into its evolution in terms of structure and attributes.

7.1.2 Network Types and Building Blocks

Any network consists of $N > 0$ nodes and $E \geq 0$ edges¹. For example, the network in [Figure 7.1](#) has $N = 34$ nodes and $E = 78$ edges.

Nodes in a network are often identified by labels (such as ``n1`` and ``n33``), which denote their unique identity. For instance, in a network representing social interactions, node labels might correspond to the names of individuals. Edges can also be labeled, though this is less common.

Attributes and Edge Types

Now, look at the network in [Figure 7.2](#) on p. 130. The network represents a form-content relationship: *Node A (an edge's origin) is linked to node B (the edge's destination) if the numerical value of B's label ("the content") equals the count of letters in A's label in the Turkish language ("the form")*. We will use it as an example of a directed complex network.

Both node and edge labels are forms of attributes. These attributes can be varied: visual (such as shape, size, color, line style, and thickness), numerical (like age or weight), or textual (including location or description). You can use attributes as dependent and independent variables in regressions and other statistical procedures.

Edges can be either directed or undirected, including those that connect a node to itself (self-loops). A directed edge has a specific orientation, indicating an asymmetric relationship between nodes, and is denoted as an arc/segment with an arrow. In [Figure 7.2](#), all edges are directed, and the edge $4 \leftrightarrow 4$ is a self-loop.

Undirected edges represent symmetric relationships where the origin of an edge is also its destination and vice versa. In a directed network, symmetric relationships can be represented by either a single undirected edge or two

¹A network with one node may have no edges.

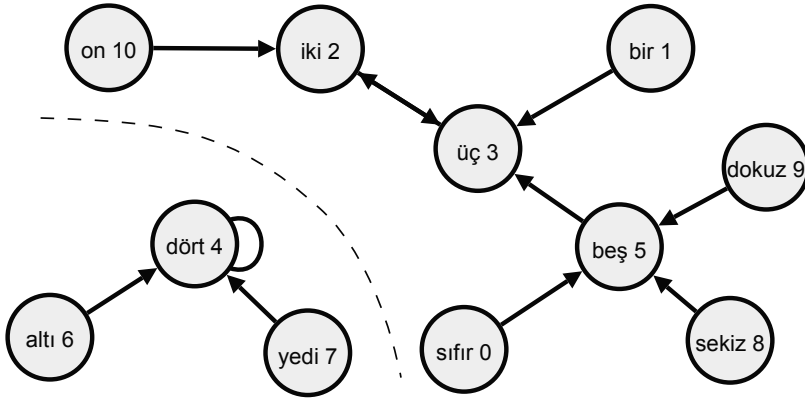


FIGURE 7.2
A form-content network of Turkish numerals.

opposite directed edges (e.g., $2 \leftrightarrow 3$, because the word “iki,” “two,” has three letters and the word “üç,” “three,” has two).

A network is classified as directed if it contains at least one directed edge. Conversely, a network with exclusively undirected edges is an undirected network. It is possible to convert an undirected network into an equivalent directed one by replacing each edge with a pair of reciprocal directed edges. However, the reverse process generally causes the loss of information.

Nodes possess attributes that may stem from external factors (exogenous attributes) or be derived from the network’s structure (endogenous attributes). One key endogenous attribute is the node’s degree d_n , which counts the number of edges incident to the node n . In directed networks, you should differentiate between indegree (number of incoming edges) and outdegree (number of outgoing edges). For example, in [Figure 7.2](#), node “beş” has a total degree of 4, with an indegree and outdegree of 3 and 1, respectively. Sometimes, nodes with an indegree of 0 are called sources (“sıfır”) and nodes with an outdegree of 0 are called sinks. There are no sinks in the provided example.

You will learn about other endogenous attributes in [Section 7.3](#) on p. 134.

Besides orientation, edges may also have weights, typically real numbers greater than 0 and not exceeding 1, assumed as 1 if not specified. Edge weight signifies the strength of the relationship, illustrated through varying line thickness². Negative weights can represent negative relationships (e.g., “Alice hates Chuck” = “Alice loves Chuck with the weight of -1”), although many CNA algorithms expect strictly positive weights.

A network with any weighted edges is termed a weighted network, where each edge is expected to have a weight, either explicitly or by default. Note

²The weight of 0 implies zero relationship strength and, therefore, the absence of an edge.

that a node degree in a weighted network becomes a weighted degree and equals the sum of the weights of all the incident edges.

When designing a network, you should carefully define node and edge attributes, including edge orientation, weightedness, and signedness. These choices may impact the data required for constructing the network.

Connectedness and Components

The degree of a node in a network reflects its immediate connectedness. Nodes with no connecting edges are identified as isolates and have a degree of zero. However, a network can still be disconnected even if it contains no isolates.

Understanding the broader connectedness and global properties of a network involves examining paths. A path between two nodes, say A and B, is a sequence of edges starting at A and ending at B, with consecutive edges connected end-to-end. A path without self-intersections is called a simple path. For instance, the path from “bir” to “üç” in [Figure 7.2](#) via “iki” and “üç” is not simple, as it intersects itself at “üç.”

Paths can vary in complexity, with some being direct and others involving multiple nodes. Another path from “bir” to “üç” consists of one direct edge; it is simple.

A cycle is a special path that starts and ends at the same node. Our example network contains multiple cycles, such as the sequence “iki” → “üç” → “iki” (two edges), “dört” → “dört” (one edge), or “dört” → “dört” → “dört” → “dört” (three edges).

In unweighted networks, the length of a path is simply the count of its comprising edges. Weighted networks might consider edge weights or other metrics to determine path lengths. For simplicity, let’s focus on unweighted networks here.

The shortest or geodesic path between two nodes is the one with the shortest length. Geodesic paths are crucial for understanding information spread in complex networks. In [Section 7.3](#) on p. 134, you will use their lengths and quantities to determine node centralities.

A connected undirected network is one where a path exists between every pair of nodes. If this condition is not met, the network can be divided into connected components: network fragments where the connectedness holds internally but not externally. An example is the road network on Earth; it is not globally connected (one cannot take an uninterrupted road trip from Boston to Melbourne), but regions like the European mainland form a connected component, ignoring the minor local roads, especially with additions like the Channel Tunnel. If the sample network in [Figure 7.2](#) were undirected, it would have two connected components separated by a dashed line.

Directed networks introduce additional complexity. Similar to one-way roads, a path from A to B does not imply a reverse path from B to A. Directed edges can be removed or treated as undirected for defining components

in directed networks. The resulting subnetworks are termed strongly or weakly connected components, respectively.

The form-content network has ten strongly connected components: a two-node component {“iki”, “üç”} and each other node as a singleton—and the same two weakly connected components as previously defined in the undirected case.

Lastly, a complex network typically has a giant connected component (GCC), which is significantly larger than others in node count (in the upper part of Figure 7.2). It is not uncommon in CNA to focus primarily on the GCC, often disregarding smaller components, especially isolates.

7.2 Forms of Network Representation

Despite their relational complexity, complex networks can be effectively represented in tabular form, primarily through an adjacency matrix or an edge list.

An adjacency matrix A is a square matrix with dimensions $N \times N$, where N is the number of nodes. The rows and columns of this matrix correspond to the network’s nodes, and the matrix entries indicate whether pairs of nodes are connected. Tables 7.1 and 7.2 illustrate examples of partial adjacency matrices for an undirected, unweighted network and a directed, possibly weighted network, respectively, based on Figures 7.1 and 7.2.

In an unweighted network, if an edge connects the node in row i to the node in column j , the matrix entry A_{ij} is set to 1; if there is no edge, A_{ij} is 0.

TABLE 7.1
Partial adjacency matrix of an undirected, unweighted network without self-loops. Cf. Figure 7.1 on p. 128.

	n0	n1	n2	n3	n4	n5	n6	...
n0	⋅	1	1	1	1	1	1	...
n1	1	⋅	1	1	0	0	0	...
n2	1	1	⋅	1	0	0	0	...
n3	1	1	1	⋅	0	0	0	...
n4	1	0	0	0	⋅	0	1	...
n5	1	0	0	0	0	⋅	1	...
n6	1	0	0	0	1	1	⋅	...
...	...	...	...	...	...	...	...	...

TABLE 7.2
Partial adjacency matrix of a directed, unweighted network with a self-loop (boxed). Cf. [Figure 7.2](#) on p. 130.

	sıfır	bir	iki	üç	dört	beş	altı	...	on
sıfır	0	0	0	0	0	1	0	...	0
bir	0	0	0	1	0	0	0	...	0
iki	0	0	0	1	0	0	0	...	0
üç	0	0	1	0	0	0	0	...	0
dört	0	0	0	0	1	0	0	...	0
beş	0	0	0	1	0	0	0	...	0
altı	0	0	0	0	1	0	0	...	0
yedi	0	0	0	0	1	0	0	...	0
sekiz	0	0	0	0	0	1	0	...	0
dokuz	0	0	0	0	0	1	0	...	0
on	0	0	1	0	0	0	0	...	0

In an undirected network, the matrix is symmetric ($A_{ij} = A_{ji}$), and networks without self-loops have a zero diagonal ($A_{ii} = 0$).

In weighted networks, the adjacency matrix contains the weights of the edges instead of just 1s and 0s. For example, a light-weighted edge might be represented as 0.5, while a thicker edge as 2.

The adjacency matrix provides an intuitive snapshot of the network’s structure. It is easily transferable to and from a CSV file (described in [Subsection 2.1.2](#) on p. 21). It is readily compatible with Python programs. However, it has some limitations:

1.

It does not accommodate additional node and edge attributes beyond edge weight.
2.

In undirected networks, the matrix’s symmetry leads to redundancy.
3.

Most elements in a typical complex network matrix are zeros, indicating no connection, which results in further matrix underutilization.

An edge list is an alternative way of representing complex networks. As the name suggests, it is a list of edges, namely, a table in which a row denotes an edge and columns denote its start and end nodes, weight, and other attributes if necessary. [Table 7.3](#) shows an edge list of the Turkish numerals diagram depicted in [Figure 7.2](#). The table is naturally suited for external storage in a CSV file.

You can easily convert an adjacency matrix into an edge list. The reverse conversion is possible, too, but with a loss of non-weight attributes.

An adjacency matrix approach is generally preferred for dense networks with many edges and no attributes. In contrast, an edge list works better for sparse networks with few edges and rich attribute structures. Since social

TABLE 7.3
Partial edge list (left) and its CSV representation (right) of a directed, unweighted network. Cf. [Figure 7.2](#).

Start	End	Weight	`Start,End,Weight`
altı	dört	1	`altı,dört,1`
beş	üç	1	`beş,üç,1`
bir	üç	1	`bir,üç,1`
dört	dört	1	`dört,dört,1`
dokuz	beş	1	`dokuz,beş,1`
iki	üç	1	`iki,üç,1`
on	iki	1	`on,iki,1`
sıfır	beş	1	`sıfır,beş,1`
sekiz	beş	1	`sekiz,beş,1`
üç	iki	1	`üç,iki,1`
yedi	dört	1	`yedi,dört,1`

networks fall into the second category, they lend themselves to the edge list representation.

Both the adjacency matrix and edge list formats are intended for the internal storage of complex networks. Externally, in addition to the previously mentioned CSV file format, you can use GraphML (an XML-based Graph Markup Language), GEXF (Graph Exchange XML Format), GML (Graph Modeling Language), DL (the native format of UCINET software), and NET (the native format of Pajek software). Python and other programming languages and tools provide extensive support for reading, writing, and conversion networks in any of these formats.

7.3 Centralities and Densities

In [Subsection 7.1.2](#), we discussed the concept of a node’s degree, an internally calculated attribute used to estimate a node’s immediate network reach. However, this degree does not fully capture a node’s long-range influence within the network. Specifically, it does not indicate how connected a node is to others in terms of sequential direct links (edges). For instance, consider nodes `n33` and `n26` in the network depicted in [Figure 7.1](#) on p. 128. Intuitively, `n33` seems more influential than `n26`, but how can we quantitatively assess this difference?

7.3.1 Closeness Centrality

Closeness centrality reflects a node’s significance in a complex network based on its average distance from the node to all other nodes. This measure is

TABLE 7.4

Five nodes with the highest closeness, degree, betweenness, and eigenvector centralities, sorted by the closeness centrality in Zachary's karate club network.

A	C_A	D_A	B_A	EV_A
n0	0.569	0.485	0.438	0.355
n2	0.559	0.303	0.144	0.317
n33	0.550	0.515	0.304	0.373
n31	0.541		0.138	
n13	0.516			
n32		0.364	0.145	0.309
n1		0.273		0.266

crucial in contexts like social networks, where the efficiency of information spread depends on intermediaries' willingness to pass on information. The more intermediaries between two points (A and B), the less likely A will receive information from B. A node should be relatively close to others in the network to be an influential spreader. Conversely, it is beneficial to distance oneself from potential infection sources in an epidemic.

Closeness centrality C_A is defined as the inverse of the average distance from a given node A to all other nodes in the network:

$$C_A = \frac{N - 1}{\sum_m l_{A,m}}. \quad (7.1)$$

Here, $l_{A,m}$ represents the shortest path length between nodes A and m . This distance is considered infinite if no path exists (indicating the nodes are in different connected components). Note that in directed networks, the distance from A to m may differ from that from m to A.

The value of C_A is always positive and does not exceed 1, reaching 1 only in a fully connected network where each node directly connects to every other node. In such networks, closeness centrality is maximized. Conversely, an isolated node (isolate) has a closeness centrality of zero.

Shorter paths from a node will decrease the denominator in Equation 7.1, thus increasing the closeness centrality value. On the other hand, longer paths will have the opposite effect. Table 7.4 lists the five nodes with the highest closeness centrality in Zachary's karate club network. Note that four of these nodes are situated along the sociogram's "backbone," stretching from the upper left to the lower right corner.

7.3.2 Degree Centrality

While there are multiple ways to measure a node's centrality in a network, one of the simplest forms is through its degree centrality. The concept is based on

the idea that a node connected to more nodes (i.e., with more edges) is better positioned for communication within the network. However, the significance of a node's connections varies depending on the network's overall size. For instance, having 17 connections in a network of 34 nodes is notably different from having the same number of connections in a network comprising 3,400 nodes. To standardize this measure across networks of various sizes, we calculate the degree centrality by dividing a node's degree by the total number of nodes in the network N :

$$D_A = \frac{d_A}{N}. \quad (7.2)$$

Degree centrality values range from 0 (representing a node with no connections) to 1 (indicating a node connected to every other node in a fully connected network). In directed networks, we distinguish between indegree and outdegree centralities, as a node might have different numbers of incoming and outgoing connections. It is important to note that for any given node A , its degree centrality D_A and closeness centrality C_A are not the same, although they are often strongly correlated.

Table 7.4 lists the five nodes with the highest degree centrality in Zachary's karate club network. Three nodes are also among those with the highest closeness centrality, demonstrating this correlation.

7.3.3 Betweenness Centrality

After discussing degree and closeness centralities, which focus on nodes' immediate connections and closeness, we now turn to betweenness centrality. This measure evaluates a node's role in controlling or influencing the flow of information (or other entities like infections in epidemic networks) through the network.

Betweenness centrality estimation begins by identifying all the shortest paths (geodesic paths) between every pair of nodes in the network. Then, for a specific node (let's call it node A), we calculate the proportion of these paths that pass through A (excluding paths that start or end at A) and average these proportions. Mathematically, if $\sigma_{mn}(A)$ denotes the number of shortest paths between nodes m and n that pass through node A , and σ_{mn} is the total number of shortest paths between m and n , the betweenness centrality for node A is given by:

$$B_A = \sum_{m \neq A \neq n} \frac{\sigma_{mn}(A)}{\sigma_{mn}} / (N-1)(N-2). \quad (7.3)$$

A node's betweenness centrality B_A ranges from 0 (for nodes not lying on any shortest paths) to 1 (for a central node in a hub-and-spoke network, where a central "hub" node is connected to multiple "spoke" nodes). Nodes with high betweenness centrality are critical for network connectivity. Their removal can

lead to significant disruption, a principle applied in strategies like contact tracing in epidemiology and intercepting messengers in counterterrorism.

7.3.4 Eigenvector Centrality

The ancient adage, often attributed to Socrates or Euripides, “Show me who your friends are, and I will tell you who you are,” aptly summarizes the essence of eigenvector centrality. This measure, the last centrality metric discussed in our text, embodies this principle by evaluating a node’s importance based on the importance of its connections.

Eigenvector centrality assesses a node’s significance not just by how many connections it has but also by the quality of these connections. If a node is connected to other highly connected nodes, it is considered more central. Mathematically, for a given node A , let O_A represent its neighborhood, which includes all the nodes adjacent to A . The eigenvector centrality of A , denoted as EV_A , is calculated as follows:

$$EV_A = \left(\sum_{n \in O_A} EV_n \right) / D_A. \quad (7.4)$$

This definition is recursive: it defines EV_A in terms of the eigenvector centralities of its neighboring nodes. Solving this equation requires knowledge of linear algebra, specifically eigenvalues and eigenvectors. However, in the context of applied Complex Network Analysis, it suffices to know that tools like Python provide built-in functions to compute eigenvector centrality for any node in a network.

7.3.5 Global and Local Density

Global and local density will give you insights into how “crowded” a network is, both as a whole and at individual node levels.

Global density, GD , measures how many edges (connections) exist in the network compared to the maximum possible number of edges. For undirected networks, the maximum number of edges $E_{\max} = N \times (N - 1) / 2$. In directed networks, each pair of nodes can be connected by two edges (one in each direction), making the maximum number of edges $E_{\max} = N \times (N - 1)$. A fully connected network, where every node is connected to every other node, is the only case where $E = E_{\max}$. The global density is the ratio of the observed number of edges (E) to the maximum possible number of edges:

$$GD = E / E_{\max}. \quad (7.5)$$

A network with no edges has a global density of zero, indicating no connections among its nodes.

While global density provides an overview of the entire network, local density, also known as the clustering coefficient, focuses on the immediate

neighborhood of individual nodes. It measures how densely connected a node's immediate neighborhood is. It is calculated by considering a subnetwork (a subset of a network) that includes a specific node A , its adjacent nodes, and all the edges among them. You then apply to this subnetwork a formula similar to that used for global density (Equation 7.5). The local density can vary significantly, especially in large and complex networks, where some nodes might be in densely connected neighborhoods while others are in sparser ones.

7.4 Network Structure Extraction

While centrality measures provide valuable insights into individual nodes within a network, they do not fully capture the intricate topological interconnectivity between these nodes. CNA offers several tools for a more in-depth structural analysis, including clique detection, core-periphery analysis, and community detection.

7.4.1 Cliques

A clique, denoted as C (or an n -clique in more specific terms), is a subnetwork of the original network in which every node within C is directly connected to every other node in C . Cliques represent the most interconnected and dense formations within a network. Typically, the nodes within a clique share common exogenous attributes, such as social status or organizational membership, which are not inherently part of the network structure but rather external characteristics.

The simplest clique is a fully connected network. Interestingly, even a single node can be considered a clique.

Figure 7.3 shows four distinct cliques ($C1$, $C2$, $C3$, and $C4$) within the Zachary Karate Club Network. Each of these cliques comprises six members, with each member forming five internal connections. A node can be part of multiple cliques (e.g., $n1$).

The identified cliques are both maximal and maximum. A maximal clique cannot be expanded by adding another node without losing its interconnectivity. A maximum clique, on the other hand, is the largest within the network. Thus, in the Zachary Karate Club Network, no cliques are larger than $C1$, $C2$, $C3$, or $C4$.

Finding perfect cliques in real-world complex networks is computationally challenging because cliques are rare. The fragility of cliques is evident, as the loss of even a single edge can disrupt the clique structure. For instance, if nodes $n7$ and $n13$ were connected in Figure 7.3, a larger seven-node clique would emerge.

An n -clique extends the concept of a full clique by allowing connections between member nodes through paths consisting of at most n edges. In our

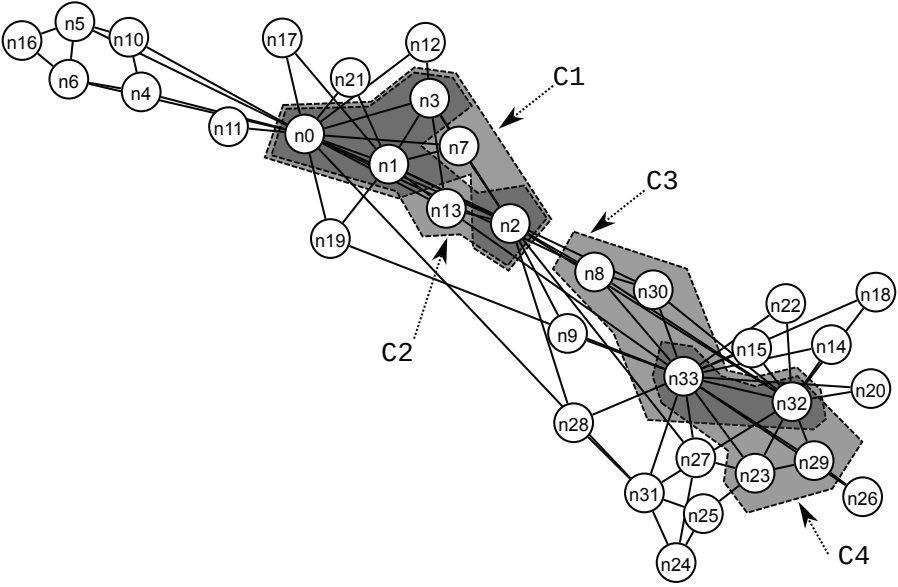


FIGURE 7.3
Maximum cliques in Zachary karate club network.

example, cliques C1, C2, C3, and C4 are also 1-cliques, while the union of C1 and C2 forms a 2-clique due to the missing edge between 'n7' and 'n13'.

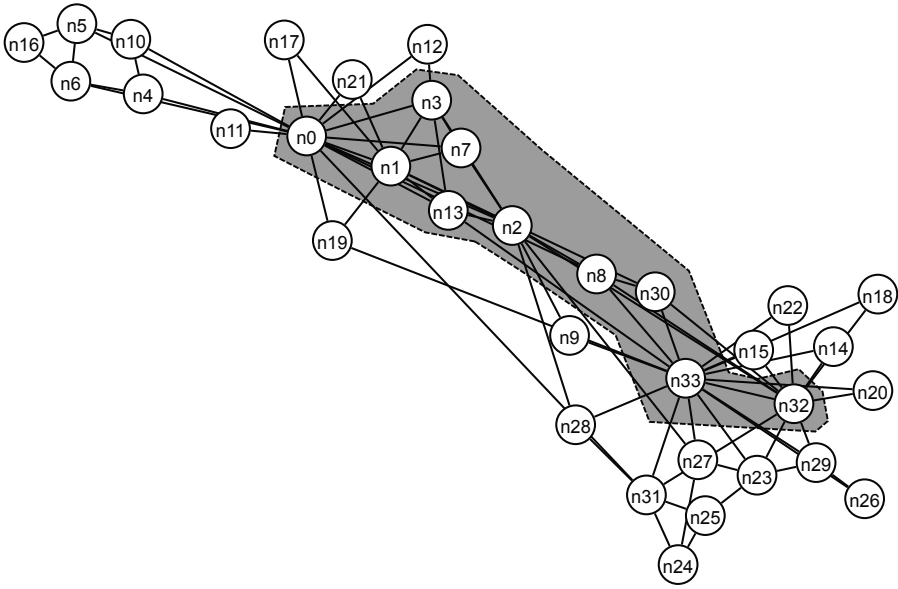
Cliques provide a foundational, bottom-up approach to analyzing network structures. This method begins with identifying structural elements like dyads (two-node cliques), triads (three-node cliques), tetrads (four-node cliques), and so on, progressively building up to maximal cliques. This approach contrasts with core-periphery analysis, which operates in a top-down manner.

7.4.2 Core-Periphery Analysis

Core-periphery analysis (CPA), also known as onion decomposition, decomposes a network into the densely connected central core and a more sparsely connected surrounding periphery consisting of coronas and crusts. CPA follows a top-down approach, starting from the broader network structure and moving toward individual components.

The process starts by systematically removing nodes with a degree less than a certain threshold k . The group of nodes removed at each step forms what is known as a corona, labeled explicitly as a k -corona based on the degree threshold.

For instance, in [Figure 7.4](#), when $k = 1$, the 1-corona comprises just one node, 'n11', directly connected only to 'n10'. This pruning process is repeated

**FIGURE 7.4**

The main core of Zachary karate club network.

until no nodes with a degree of k remain. Subsequently, the remaining nodes form the 2-core of the network, indicating they all have a degree of at least 2.

Continuing this “peeling” process for nodes with a degree of $k = 2$ results in the formation of a 2-corona, consisting of nodes $\{`n20`, `n22`, `n26`, `n12`, `12`, `n21`, `n16`, `n9`, `n15`, `n17`, `n18`, `n14`\}$. The nodes that survive this stage form the 3-core, having a degree of at least $k = 3$. The peeling continues until CPA identifies the main core of the network, which includes nodes with the highest degree, as highlighted in [Figure 7.4](#).

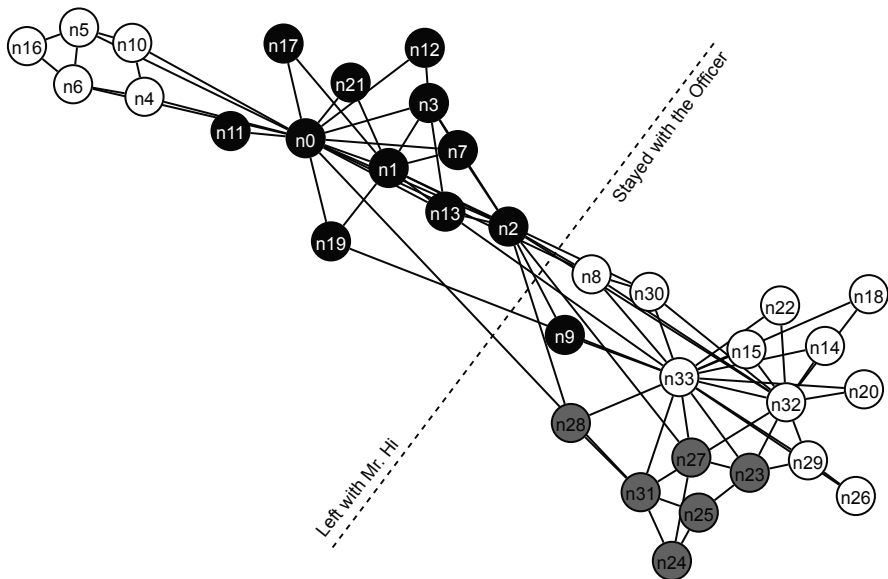
In addition to coronas and cores, CPA also defines the concept of crusts. A k -crust is the cumulative union of all coronas with a degree equal to or less than k . Therefore, for any chosen k , a network can be effectively segmented into the k -core and its complementary k -crust.

It is important to note that nodes within a core, even the main core, do not necessarily have to be fully interconnected.

7.4.3 Community Detection

While cliques, cores, coronas, and crusts provide mathematically rigorous structures within graph theory, they may not always align with the analytical needs of social science research. An alternative approach, particularly suitable for social networks, is modularity-based community detection.

Community detection involves partitioning a network into clusters or communities of nodes. The challenge lies in the many possible ways to partition a

**FIGURE 7.5**

Four detected communities in Zachary karate club network.

network. Not every partition accurately reflects the network's structure of interconnections. A measure like modularity is essential to evaluate and compare different partitions.

As defined in contemporary network theory [New06], modularity quantifies intra-community cohesion and inter-community coupling. Modularity ranges from -0.5 to 1.0. A high modularity score indicates strong cohesion within communities (most edges in a group lead to nodes within the same group) and weak coupling between communities (few edges link different groups). High modularity signifies a clear, distinct community structure. Conversely, low modularity indicates less distinct communities with more inter-group connections, suggesting a more chaotic partition.

Community detection algorithms, like the renowned Louvain algorithm [BGLL08], iteratively compare partitions to find the one with the highest modularity. The process continues until improvements in modularity become negligible.

Figure 7.5 shows the karate club network partitioned into two communities. The low ratio of edges that cross between communities to the total number of edges indicates a distinct separation of the communities.

Modularity-based community detection is advantageous for its flexibility and a natural fit for many networks. Unlike some clustering methods that require pre-specifying the number of clusters, this approach dynamically determines the number of communities to optimize modularity. However, it is

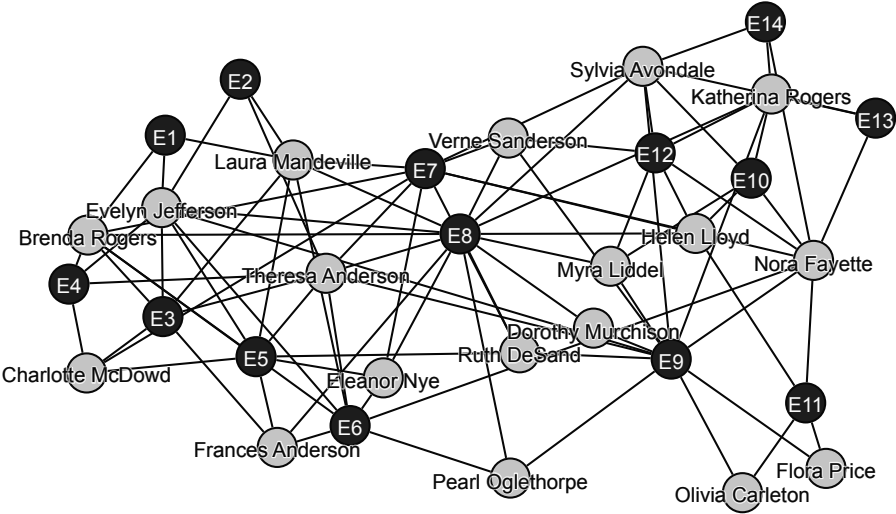


FIGURE 7.6
Unweighted bipartite Davis Southern women event network. Node color denotes the two parts.

essential to note the limitations. Community detection algorithms, including the Louvain method, can yield different results upon multiple executions, affecting reproducibility. Moreover, these algorithms sometimes struggle with resolution issues, failing to identify overly large or overly small communities. This limitation can be partially addressed by applying community detection recursively to previously identified subcomponents.

7.5 Bipartite Networks

Until now, you have seen networks where all nodes fall into a single category. However, nodes can belong to different classes. For instance, organizational charts often include both organizations and individuals. When a network consists of two distinct types of nodes, it is called a bipartite network. Networks with three or more types of nodes, known as tri-partite or higher-partite networks, also exist but are not covered in this text.

A classic example of a bipartite network is the “Southern Women” dataset, introduced by Allison Davis, Burleigh Gardner, and Mary Gardner in “Deep South” [DGG41]. This network maps the interactions of 18 women attending 14 social events over nine months in the 1930s Southern United States. The network is directional and unweighted, with edges representing women’s event co-attendance (as shown in Figure 7.6 on p. 142).

While a bipartite network shares many properties with complex networks, such as exogenous and endogenous attributes and community structure, some properties are less applicable or irrelevant in a bipartite context. Researchers often transform bipartite networks into unipartite networks through a process called projection. Projection involves retaining nodes of one type and replacing nodes of the other with edges.

In the projection process, two nodes of the same type (e.g., two women) are linked if they share connections to the same nodes of the other type (e.g., attending the same event) in the original network. The number of these shared connections determines the weight of the new edge. For instance, if Evelyn Jefferson and Laura Mandeville co-attended four events (E1, E2, E3, and E5), they would be connected by an edge with a weight of 4 in the projected network. Thus, the resulting network is weighted, but the weights can be ignored based on the analysis requirements.

Figure 7.7 on p. 143 displays a weighted projected Southern Women dataset network. It is important to note that projection is a one-way process: the projected network loses specific information from the original bipartite network (e.g., event details in a women-only projection).

Once projected, a bipartite network becomes indistinguishable from a unipartite network and can be analyzed using standard network analysis techniques.

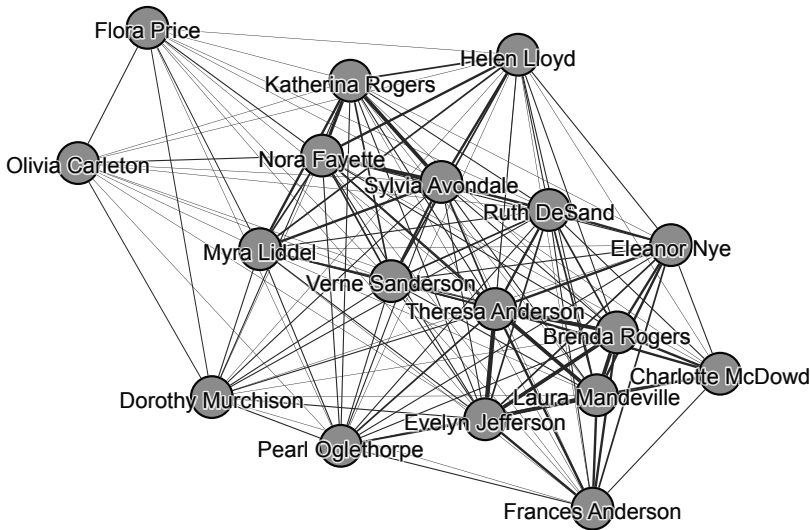


FIGURE 7.7

Weighted projected Davis Southern women event network (the attendants view); cf. Figure 7.7 on p. 143.

7.6 SNA as a Statistical Tool

In addition to offering a comprehensive understanding of complex networks, Complex Network Analysis (CNA) and Social Network Analysis (SNA) provide a range of variables for statistical analysis.

Endogenous node attributes, discussed in [Section 7.3](#) on p. 134, form a set of continuous variables. Due to their hierarchical nature, other structural features like core, corona, and crust numbers are ordinal variables. Attributes such as clique memberships and community affiliations function as categorical variables.

When combined with specific exogenous node attributes (such as demographic details in social networks, part-of-speech tags in linguistic networks, or ownership types in corporate networks), they can be utilized as dependent or independent variables in various statistical methods, including regression models and classification techniques.

To illustrate the application of Ordinary Least Squares regression in network analysis, let's revisit the karate club case (as depicted in [Figure 7.5](#) on p. 141). In this example, we use clustering coefficients, betweenness centrality, and the identified communities as endogenous attributes. The dependent variable is based on actual data indicating whether club members remained with the club president (referred to as “the Officer”) or chose to follow the instructor (known as “Mr. Hi”).

The results of the regression analysis are presented in [Table 7.5](#). In this analysis, the node-level continuous variables (such as betweenness centrality) and the categorical community identifiers show statistical significance.

What stands out in this analysis is the accuracy of the prediction, considering the club's choice is an external variable unknown to the Social Network Analysis algorithms, underscoring the effectiveness of combining various network attributes in regression models to predict outcomes within social networks.

7.7 What's Next

Text documents written in natural languages, such as English, are valuable for constructing social networks and providing independent information sources. In the next chapter, you will learn to work with texts, including techniques for extracting relevant data from text, such as identifying key entities and their relationships.

TABLE 7.5
Logistic regression analysis: determining factors influencing karate club members’ decision to stay with the officer or leave with Mr. Hi

Model:	OLS	Adj. R-squared:	0.829
Dependent Variable:	officer	AIC:	-3.4474
Date:	2023-11-30 02:40	BIC:	7.2371
No. Observations:	34	Log-Likelihood:	8.7237
Df Model:	6	F-statistic:	27.60
Df Residuals:	27	Prob (F-statistic):	2.57e-10
R-squared:	0.860	Scale:	0.044135

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
Intercept	0.4004	0.1225	3.2691	0.0029	0.1491	0.6516
C(comm)[T.1]	0.8158	0.0919	8.8772	0.0000	0.6272	1.0044
C(comm)[T.2]	0.7652	0.1160	6.5951	0.0000	0.5271	1.0032
C(comm)[T.3]	-0.0913	0.1137	-0.8030	0.4290	-0.3245	0.1420
betw	2.2956	0.9562	2.4008	0.0235	0.3337	4.2575
clus	-0.3086	0.1446	-2.1340	0.0421	-0.6052	-0.0119
betw:clus	-21.3248	6.1228	-3.4829	0.0017	-33.8878	-8.7619

Omnibus:	9.650	Durbin-Watson:	2.532
Prob(Omnibus):	0.008	Jarque-Bera (JB):	21.213
Skew:	-0.180	Prob(JB):	0.000
Kurtosis:	6.853	Condition No.:	212

7.8 Discussion Questions

1. Is it possible to have a network with one or more edges but zero nodes?
2. Argue that an isolate is both a weakly and a strongly connected component.
3. How do different centrality measures (degree, closeness, betweenness, and eigenvector) provide unique insights into the roles and influence of nodes in a network? Discuss a scenario where one centrality measure might be more informative than others.
4. What are the potential benefits and drawbacks of projecting a bipartite network into a unipartite one? Consider how certain information or structural nuances might be lost or misrepresented in the projection process.

5. Community detection is essential for understanding the structure of complex networks. What challenges are faced in accurately detecting communities within a network, and how might these challenges be addressed? Discuss the role of algorithms like the Louvain method in this context and consider their limitations.

Natural Language Processing (NLP) is concerned with techniques for deconstructing, manipulating (including translating), and generating texts in natural (human) languages, such as English or Chinese¹. NLP algorithms and tools date back to the 1950s but experienced an explosive expansion in the mid-2000s (e.g., Google Translate). NLP is the foundation of text mining, which involves extracting, organizing, and statistically analyzing text data.

Since this book is about statistical analysis, starting with a conversation about NLP is unavoidable. Once you familiarize yourself with the NLP text processing pipeline, you will learn about its applications in statistics.

8.1 Text Processing Pipeline

The Text Processing Pipeline (TPP) forms the backbone of NLP. It is a series of steps that transform text into structured data, segmenting it into sentences, words, and tokens and enriching it with linguistic attributes like part-of-speech tags. While there is no “one-size-fits-all” TPP, it generally includes:

- Denoising.
- Sentence and Word Tokenization.
- Part-of-speech Tagging.
- Stemming and Lemmatization.
- Named Entity Recognition.
- Stop Word Removal.

The rest of the chapter covers these operations and more. But first, you need to recognize the importance of language detection and examine the role of corpora in NLP.

¹It is a good place to remind you that Python is not a natural language but is considered the most natural of the existing programming languages.

8.1.1 Denoising

Analyzing documents with plain text, free of metadata like formatting tags and attributes, simplifies the process. Consider a blog post from LiveJournal, a once-popular blogging website, made in 2005 by an anonymous user known as *eggy_the_elf*:

```
<html>Look what I learned in therapy today mommy!,I had no clue
  I could be my own person...  <br> <br> <span
    style='text-decoration: underline;*>I am</span> <span
    style='font-weight: bold;*>upset</span> because I can't
speak my mind. <br> <br> <span style='text-decoration:
underline;*>I am</span> <span style='font-weight:
bold;*>angry</span> because I can't voice what I want heard.
<br> <br> <span style='text-decoration: underline;*>I
am</span> <span style='font-weight: bold;*>sad</span>
because I feel helpless. <br> <br> I am <span
style='font-weight: bold;*>hurt</span> because I ruined my
life & stuff.  <br> <br> <br> that's absolutely
amazing!!
</html>
```

The document is laden with HTML tags and entities: text fragments that either represent otherwise valid characters (e.g., `&` stands for `&` and ` ` is an unbreakable space) or control the way valid characters are displayed (`<span style='font-weight: bold;\*>X</span>` shows **X** in bold font). These elements do not contribute to the text's literal meaning and may be removed or replaced before processing. The cleaned text then appears devoid of such metadata:

```
Look what I learned in therapy today mommy!,I had no clue I
could be my own person... I am upset because I can't speak my
mind. I am angry because I can't voice what I want heard. I am
sad because I feel helpless. I am hurt because I ruined my
life & stuff. that's absolutely amazing!!
```

But is it a text in English?

8.1.2 Language Detection

Many NLP algorithms are language-aware. For example, word tokenization in English depends on the definition of a word as a sequence of alphabetic characters separated by non-alphabetic characters. However, this definition is not applicable in Chinese. In Chinese, word segmentation is a complex process due to the absence of spaces between words, but the language does indeed have a concept of words, contrary to the misconception that it does not. Different languages have different capitalization rules (no capital letters in Arabic!), morphology (such as word stems, prefixes, suffixes, and endings),

syntax (sentence structure), and stop words (common words often filtered out in language processing due to their low informational content).

To summarize, you must know the language of the document to apply NLP successfully. If you work with collections of documents written in multiple languages, you must separate the documents by their language. A small text fragment's language (a few words or sentences) cannot be detected reliably, especially if several closely related languages are involved. For example, the sentence "Hand in Hand" is written the same way (and means the same thing) in English and German.

Language detection of a longer text usually yields better results. Assuming you expect the text to be in English, count how many words in the text are also in an English dictionary. If the frequency of English words in the text is high, whatever it means, the text is likely to be in English. If you can access many single-language dictionaries, you can detect the language by choosing the highest frequency of in-dictionary words.

The proposed approach, while simple, relies on a "chicken-and-egg" contradiction. You need to know the document's language to tokenize it into words, yet you need to know the words to look them up in a dictionary to detect the language. The problem can be solved by avoiding tokenization and analyzing character n -grams (sequences of n characters with no implied word breaks). You can also employ iterative refinement (assume that the text is written in language $\mathcal{L}$; tokenize; calculate the frequency of words in the dictionary for $\mathcal{L}$; ensure that it is high enough to confirm that the language is indeed $\mathcal{L}$). Other language detection algorithms fall outside of this book's scope.

8.1.3 Text Corpora

You need a language dictionary in a digital form to perform as described above. Dictionaries and other consistently composed and often standardized sets of texts in NLP are called corpora (singular: "corpus"). Corpora differ in purpose (dictionaries, training/teaching texts, texts by genre, synthetic texts), size, diversity of languages (including parallel corpora with the same text in multiple languages), specialization (legal, medical), level of annotation, and other properties.

A corpus is a foundational resource in NLP, enabling the study of language and the development of computational models that understand and generate human language. Some examples of NLP corpora include:

- WordNet: A lexical database that organizes words into sets of synonymous words (synsets) and shows the semantic connections between words.
- PennTreebank: An extensive collection of sentences from the Wall Street Journal and similar sources.
- Brown Corpus: A standard corpus of present-day edited American English.
- COCA: The corpus of contemporary American English with 365 million words.

- Flickr 8k Dataset: 8,000 images with descriptions.

The latter example is not purely textual, illustrating that corpora could be used in cross-discipline tasks not limited to text processing.

Many NLP corpora can be downloaded for free or are included in software tool distributions. For example, the Natural Language Toolkit (NLTK) contains a corpus of English words with 235 thousand entries. According to that corpus, the frequency of English words in the previously quoted LiveJournal text is 97%. The text is definitely written in English.

8.1.4 Tokenization

Tokenization is segmenting a line of characters (or, generally speaking, any symbols) into meaningful fragments. NLP is concerned with several types of tokenization, including sentence tokenization, which converts a block of text into sentences, and word tokenization, which breaks down sentences into words. You can apply word tokenization directly to the complete text to produce words but bypass the sentences.

Sentence Tokenization

Sentence tokenization relies on defining what constitutes a sentence. This definition is not uniform globally, as different languages have their own rules. Typically, modern languages utilize punctuation marks such as periods, exclamation points, question marks, and ellipses, along with their combinations, to indicate the end of a sentence. However, this process becomes complex with compound sentences, like direct quotations. These sentences may contain parts that are complete sentences themselves, leading to a dilemma: Should you treat the entire compound sentence as one unit or as multiple sentences? For instance, in such scenarios, an exclamation point might not always signal the end of a sentence; it could merely mark the end of a part within the compound sentence. This ambiguity and unorthodox punctuation (“mommy ! I had”) add to the challenges of accurately determining sentence boundaries.

Practical sentence tokenizers (software tools that execute text segmentation) use reasonable heuristics² to balance segmentation quality and speed. They are not perfect, as you can see from the example below:

1. Look what I learned in therapy today mommy!,I had no clue I could be my own person...
2. I am upset because I can't speak my mind.
3. I am angry because I can't voice what I want heard.
4. I am sad because I feel helpless.

²Heuristics are methods that help problem-solving and decision-making but do not guarantee perfect results.

5. I am hurt because I ruined my life & stuff.
6. that's absolutely amazing!
7. !

The output is mostly correct (see the original text on p. 148). However, the first two sentences are “under-fragmented” (because of the odd combination of an exclamation point and a comma), and the last one is “over-fragmented” (because of two exclamation points in a row).

Word Tokenization

Word tokenization, the process of dividing a text into words, depends heavily on how a word is defined. This definition is not as straightforward as it might seem. While it is common for words in all languages to consist of letters, many languages also incorporate other characters in their words: hyphens, apostrophes, ligatures, slashes (like in “and/or”), and even ampersands.

Moreover, words are not separated by spaces in several languages, such as Chinese, Japanese, Thai, Tibetan, Lao, Khmer, and Burmese. This characteristic presents a unique challenge for word tokenization in these languages, as standard methods relying on spaces to identify word boundaries become ineffective.

Additionally, compound words in some languages may include numbers, and it is not uncommon for abbreviations to end with a period. These variations in word composition across languages highlight the complexity of accurately identifying word boundaries in natural language processing.

The solution used in typical NLP tasks is again based on language-specific heuristics. Here are three examples of segmenting the second sentence with three different word tokenizers (the vertical bars mark word boundaries):

1. I|am|upset|because|I|ca|n't|speak|my|mind|.
2. I|am|upset|because|I|can|'|t|speak|my|mind|.
3. I|am|upset|because|I|can't|speak|my|mind|.

The fragment that causes the difference is the contraction “can’t.” The first tokenizer treats it as two words, “ca” and “n’t,” presumably as a tribute to “not.” The second prefers to treat the apostrophe as a separate third word. The third tokenizer conveniently recognizes the contraction as one word.

The choice of tokenizer affects the perceived correctness of the segmentation; what you consider correct can vary based on the specific requirements of the NLP task. Additionally, it is noteworthy that all three tokenizers treat the sentence-terminating period as a separate word. This example underscores the complexity and variability inherent in word tokenization in NLP.

8.1.5 Part-of-Speech Tagging

After the word tokenization stage, which produces a list of tokens comprising punctuation and words, the next step is to identify the part-of-speech (PoS) for each word. This process, known as PoS tagging, determines whether a word functions as a noun, verb, adjective, adverb, or another grammatical category. This information is crucial for effective stemming and lemmatization (see [Subsection 8.1.7](#)) and more advanced semantic analysis.

Like tokenization, PoS tagging is highly language-specific and can be pretty complex. For instance, in English, while certain word forms or endings might hint at specific parts of speech (e.g., “-ly” for adverbs, “-tion” for nouns, and “-ing” for present particles and gerunds), there is no universally reliable algorithm for accurately identifying the PoS in every case. Languages with fixed word orders, such as Mandarin Chinese and Swahili (Subject-Verb-Object), Korean (Subject-Object-Verb), Malagasy (Verb-Object-Subject), and Xavante (Object-Subject-Verb), offer some clues for PoS tagging, but the task remains challenging due to linguistic nuances.

PoS tagging relies heavily on heuristics and is complicated by the inherent ambiguities and exceptions in natural languages. A famous example that demonstrates this complexity is the sentence “Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo.” This sentence is grammatically valid but highly ambiguous in English, often causing standard PoS taggers, like the one in the Natural Language Toolkit (NLTK), to struggle with accurate classification. (NLTK PoS tagger claims that all “buffalos” are nouns.)

However, PoS taggers generally perform much better with simpler and more conventional texts. For example, consider the already familiar tagged sentence from the LiveJournal post, where words are categorized as pronouns (PRP, PRP\$), verbs (VB, VBP), nouns (NN), adjectives (JJ), and punctuation:

1. ...
2. ...
3. I^{PRP} | am^{VBP} | upset^{JJ} | ... | can't^{VBP} | speak^{VB} | my^{PRP\$} | mind^{NN} | .

In this example, punctuation like the period is tagged appropriately, not as a separate part-of-speech but as a punctuation mark.

Understanding the PoS of words enables further processing, such as reducing them to their dictionary forms, a key step in many NLP applications.

8.1.6 Stop Word Removal

In many languages, certain words frequently occur but convey relatively little meaning. For instance, articles like “a/an” and “the” indicate whether a noun is previously mentioned, but their omission often leaves text still understandable. Similar is the case with conjunctions, prepositions, and auxiliary verbs

such as “to be” and “to have,” along with certain pronouns. In contrast, removing more meaningful words like “auxiliary” or “prepositions” could lead to significant misunderstanding.

These high-frequency yet low-information words are known as stop words. Removing stop words is common in tasks where analyzing word frequencies, such as text summarization, is essential. Various languages have lists of stop words, which can differ in size and composition. For example, standard English, Turkish, Danish, Indonesian, and Slovene stop lists contain 180, 53, 94, 758, and 1,784 words respectively.

It is important to note that these standard stop lists are not universally applicable. They are effective on average but might not suit specific datasets or analysis goals. For instance, if your study focuses on using personal pronouns, removing them as stop words would be counterproductive. Similarly, in specialized fields like medical or legal texts, you might consider most non-subject-specific words as stop words.

Here is how *eggy_the_elf*'s post may look after “standard” stop word and punctuation elimination and further conversion to the lowercase:

```
look learned therapy today mommy clue could person upset speak  
mind angry voice want heard sad feel helpless hurt ruined life  
stuff absolutely amazing
```

Although this processed text is ungrammatical and somewhat ambiguous, it generally retains the core information about the conversation topic. However, be mindful that the effectiveness of stop word removal can vary based on the context and purpose of the text analysis.

8.1.7 Stemming and Lemmatization

In the Text Processing Pipeline (TPP), stemming and lemmatization are crucial, though not always the final stages. These processes aim to simplify words to a more basic form, but they do so differently.

Stemming and lemmatization reduce various grammatical forms of a word to a single token. Lemmatization transforms a word into a lemma—a dictionary form of the word while stemming may result in a form that is not necessarily a recognized dictionary word. Lemmatizers and stemmers are the tools used for these tasks, respectively.

For example, in lemmatization, verbs “are,” “is,” and “was” are all reduced to “be,” and the plural noun “women” becomes the singular “woman.” In contrast, stemming might produce fragments like “ar” and “wa” from these words, which are not standard English words.

The differences between these approaches are illustrated in [Table 8.1](#), where results from two popular stemmers (Lancaster³ and Porter⁴) are com-

³Developed by Chris Paice from Lancaster University.

⁴Named after its author, Martin Porter.

TABLE 8.1
Lemmatization and stemming results for select
English words

Word	Lemma	Lancaster	Porter
are	be	*ar	?are
was	be	?was	*wa
women	woman	?women	*wom
honestly	honestly	?honest	*honestli
entities	entity	*ent	*entiti
magically	magically	*mag	?magic
ideas	idea	idea	idea

pared against lemmatization results. The table uses asterisks (*) to denote ungrammatical results and question marks (?) for technically correct but incomplete results.

While stemmers often make more errors than lemmatizers, they are still widely used. Stemmers are faster, use simpler algorithms, and do not require additional grammatical information like part-of-speech. They are instrumental in tasks where speed and efficiency are more critical than absolute accuracy.

However, the choice between stemmers and lemmatizers depends on the specific requirements of the NLP task. For instance, lemmatizers are preferable in tasks that require a high degree of linguistic accuracy, such as semantic analysis. On the other hand, stemmers are often more practical for large-scale text processing where speed is a priority.

To illustrate the impact of stemming on text, here are examples of the LiveJournal post processed by Lancaster and Porter stemmers (with stop words and punctuation removed) and converted to lowercase.

Lancaster Stemmer:

look learn therapy today mommy clu could person upset speak mind
angry voic want heard sad feel helpless hurt ruin lif stuff
absolv amaz

Porter stemmer:

look learn therapi today mommi clue could person upset speak mind
angri voic want heard sad feel helpless hurt ruin life stuff
absolut amaz

These processed texts appear grammatically incorrect and un-English. Yet, they retain essential information for statistical analysis or other NLP tasks where the exact form of words is less important than their root meaning or frequency.

TABLE 8.2
The most frequently used n-grams in the Declaration of Independence

Words	“State” (2.75%), “law” (2.48%), “government” (2.20%), “power” (1.66%), “right” (1.38%), “colony” (1.10%), “assent” (1.10%)
Bigrams	“Independent state” (0.83%), “state america” (0.55%), “great britain” (0.55%), “assent law” (0.55%), “war peace” (0.55%), “free independent” (0.55%)
Trigrams	“Free independent state” (0.55%)

8.2 Frequency Analysis

The text processing pipeline, while a powerful tool by itself, is usually just a precursor to statistical analysis tasks. The simplest of them is frequency analysis, which is the calculation and further comparison of frequencies of occurrences of words in a text or multiple texts. Frequency analysis is a stepping stone towards summarization (Section 8.3 on p. 156) and sentiment analysis (Section 8.5 on p. 160).

The frequency f_W of a word W in a text T consisting of n_T words is simply the number n_W of occurrences of W in T normalized (divided) by the text size:

$$f_W = \frac{n_W}{n_T}, 0 \leq f_W \leq 1.$$

Here, a “word” is any token produced by a word tokenizer (including punctuation) or any larger syntactic unit, such as a bigram (a pair of consecutive words), a trigram (three consecutive words), or n-gram (n consecutive words). Consequently, frequency analysis can be applied to words, bigrams, trigrams, and n-grams.

Frequency analysis of small text fragments (such as *eggy_the_elf*’s blog post) does not produce exciting results because most words in a small text are unique. Let’s apply the technique to a longer document, the Declaration of Independence of the USA⁵ (1776). The Declaration has 363 tokens (after the elimination of stop words and punctuation).

Table 8.2 shows the results of the frequency analysis. Note that the trigram “free independent state” and bigrams “free independent” and “independent state” are repeated twice each, meaning that the trigram is the result of overlapping of the bigrams.

Even the shallow analysis of the words in the table suggests that the document refers to politics, specifically to the government of a country that either has colonies or is/was a colony. The bigrams and the trigram clarify that there

⁵<https://www.archives.gov/founding-docs/declaration-transcript>

are two countries: “state[s of] America” and “Great Britain”—and they are engaged in a war for independence (it is still not clear which country is the colonial power).

To summarize, vanilla frequency analysis can be used as a trivial summarization tool—but arguably should not be because there are better alternatives.

8.3 Summarization

Text summarization shortens long text pieces while preserving key information content and overall meaning to create a summary. It assists news aggregation and content curation.

There are two main types of text summarization techniques: extractive and abstractive.

8.3.1 Extractive Summarization

Extractive summarization selects the most informative phrases from the original text. In essence, it extends the summarization by frequency technique (Section 8.2 on p. 155): it starts with ranging all lemmas in the text by their frequency and then computes sentence scores by summing frequencies in each sentence. The sentences with the highest scores are selected for the summary (“extracted”), while others are ignored.

As an example of a practical extractive summarization, let us summarize the content of this whole chapter⁶. There are approximately 900 distinct lemmas in the chapter. The top ten are “word” (89), “text” (59), “language” (40), “sentence” (28), “NLP” (20), “English” (18), “analysis” (18), “name” (17), “entity” (16), and “example” (16). The numbers in the parentheses are the raw frequencies before normalization. You do not need to normalize them because dividing frequencies by the total text size does not change their order.

Here is the list of the seven most informative sentences of the chapter in their original order. The ten most frequent (therefore, most important) words of the text are highlighted in each sentence:

In Chinese, **word** segmentation is a complex process due to the absence of spaces between **words**, but the **language** does indeed have a concept of **words**, contrary to the misconception that it does not. Different **languages** have different capitalization rules (no capital letters in Arabic!), morphology (such as **word** stems, prefixes, suffixes, and endings), syntax (**sentence** structure), and stop **words** (common **words** often filtered out

⁶Due to the ongoing editing process, the text quoted in this section may differ from the original text.

in **language** processing due to their low informational content). Corpora differ in purpose (dictionaries, training/teaching **texts**, **texts** by genre, synthetic **texts**), size, diversity of **languages** (including parallel corpora with the same **text** in multiple **languages**), specialization (legal, medical), level of annotation, and other properties. **NLP** is concerned with two types of tokenization: **sentence** tokenization, which converts a block of **text** into **sentences**, and **word** tokenization, which breaks down **sentences** into **words**. **Word** tokenization, the process of dividing a **text** into **words**, depends heavily on how a **word** is defined. Lemmatization transforms a **word** into a lemma—a dictionary form of the **word** while stemming may result in a form that is not necessarily a recognized dictionary **word**. Here, a “**word**” is any token produced by a **word** tokenizer (including punctuation) or any larger syntactic unit, such as a bigram (a pair of consecutive **words**), a trigram (three consecutive **words**), or n-gram (n consecutive **words**).

Despite being fast and intuitive, extractive summarization has several major drawbacks.

- Picking the most informative sentences almost randomly creates an incoherent summary that is hard to follow.
- Similar sentences may have similar scores and be extracted together, introducing redundancy.
- The method is incapable of generating new sentences and creating abstract representations.
- In long texts with many contexts, extractive summarization may struggle to identify the relevant keywords.
- It quickly falls victim to frequency bias, picking up the words that are used unjustifiably too frequently.
- The quality of the summary depends on the quality of the original text.

The alternative class of summarization techniques, abstractive summarization, is free of these issues.

8.3.2 Abstractive Summarization

Abstractive summarization creates new phrases that express the main ideas in a condensed form. It produces better results than extractive summarization but requires more computational resources, generates less reliable summaries, and often requires human oversight.

Abstractive summarization is typically based on transformers, such as the Generative Pre-trained Transformer (GPT) from OpenAI or transformers from Hugging Face or TensorFlow.

A transformer is a machine-learning model (specifically, a deep-learning model) that extracts the words' context in addition to words and their lemmas. Transformers look at the entire sequence and understand how each word relates to all the other words in the sentence. For example, a transformer “recognizes” the difference between the word “bank” in “river bank” and “savings bank” based on the context. Finally, the attention mechanism allows a transformer to focus on different text parts when producing the output rather than reading the text sequentially. By understanding word nuances and relationships, transformers can elevate the content of a document to a more abstract level and eventually express it as a summary.

Further discussion of transformers is outside this book's scope. However, you may want to enjoy the summary of this chapter prepared by GPT4.

Natural Language Processing (**NLP**) involves the manipulation and understanding of human language by machines, serving as the basis for **text** mining and **analysis**. The **Text** Processing Pipeline (TPP) is essential in **NLP**, transforming **text** into structured data through several steps like tokenization, tagging, and stemming. The chapter emphasizes the importance of **language** detection and the role of corpora in **NLP**, highlighting that accurate **language** detection is crucial for applying **NLP** techniques effectively. Corpora, structured sets of **texts** like dictionaries and databases, are fundamental resources in **NLP** for **language** study and model development. The chapter discusses tokenization, the segmentation of **text** into meaningful units, which varies significantly across **languages** due to different definitions of what constitutes a **word** or **sentence**. It also covers part of speech tagging, a **language**-specific process that is essential for further linguistic **analysis**, despite the complexities and ambiguities inherent in natural **languages**. Lastly, the **text** delves into frequency **analysis**, stemming and lemmatization, sentiment **analysis**, and **named entity** recognition, illustrating the advanced tools and methods used in **NLP** to extract and analyze information from **text** data.

Discussion question 4 on p. 163 compares the two summarization mechanisms.

8.4 Named Entities Recognition

The responsibility of named entity recognition (NER) is to identify and classify named entities.

A named entity is an object denoted with a proper name. Named entities include person names and nicknames (“Peter,” “eggy_the_elf”), geographical locations (“Boston,” “Massachusetts,” “United States of America”), organizations (“Suffolk University”), brand and product names (“Excel,”

“Campbell”), nationalities (“Apache,” “French”), monetary values (“\$10,000”), measurement units (“meter,” “Joule”), time expression (“June”), and so on—all proper names and more.

The definition of a named entity is flexible and often depends on the purpose of the study. `SpaCy`, a natural language processing library written in Python, reports eighteen classes of named entities. If you analyze the co-mentioning of actors in a text (say, to construct a complex social network, see [Chapter 7](#) on p. 127), then it suffices to recognize person names. Analyzing a financial report requires knowing company names, stock exchange symbols, and monetary values.

NER is an empiric process based on best practices (it goes without saying that it is also language-specific). It may misrecognize a named entity or miss it. It may also classify a named entity into an incorrect category. You have to check the output of a NER engine before using it in a critical application—or rely on a consensus-based or majority-based output of more than one engine.

There are three most common classes of NER systems:

- Supervised machine learning-based (ML) systems use machine learning models trained on texts annotated by human coders. The coders identify and mark named entities. Then, an ML algorithm extracts typical patterns that the entities match and applies those patterns to new texts. ML models require large volumes of training data.
- In rules-based systems, the patterns (known as rules), such as capitalizations or titles, are not discovered automatically by training but provided by programmers. Such systems can easily miss named entities that do not follow the *a priori* rules and are inferior to the ML systems in handling complexity.
- Finally, dictionary-based systems rely on pre-constructed dictionaries of named entities (such as the `names` corpus from NLTK containing 8,000 English male and female names; see [Subsection 8.1.3](#) on p. 149). Such systems fail to extract named entities not listed in the dictionaries.

Let’s extract named entities from a paragraph on p. 25 of this book as an example of NER in action:

For instance, gender may be encoded as follows: male = 0 and female = 1. In the case of ethnicity, the assignment might look like this: white = 1, Black or African American = 2, Hispanic/Latino = 3, Asian = 4, American Indian/Alaska Native = 5, Native Hawaiian and Other Pacific Islanders = 6, and Multiracial = 7.

SpaCy reports 0, 1, 1, 2, 3, 4, 5, 6, and 7 as cardinal numbers, “African American,” “Hispanic,” “Asian,” “American,” “Indian”, “Native Hawaiian,” and “Pacific Islander” as NORP (“Nationalities or religious or political groups”), and “Alaskan Native” as a Person. It remains unclear why the tool

separated “Asian” and “American” but kept “African American” together. Tagging “Alaskan Native” as a Person is also debatable but not entirely incorrect.

8.5 Sentiment Analysis

Is the document you want to analyze good or bad? Funny or sad? Sarcastic or humorous? Enters sentiment analysis: the process of determining the emotional tone of a text. Sentiment analysis is broadly used to recognize, quantify, and categorize opinions, popularity, and other subjective information. The most common scale used by sentiment analyzers is “negative”–“neutral”–“positive,” but other subject-specific scales are also possible.

One of the most popular standard sentiment analyzers is VADER (Valence Aware Dictionary and sEntiment Reasoner). VADER was specifically trained to analyze social media posts (and even more specifically—posts on `x`, formerly known as Twitter). Such posts are often emotionally charged and consist of short sentences with an abundance of abbreviations and slang words. VADER does not work well with longer, well-structured texts, but it naturally applies to LiveJournal posts.

VADER reports four scores. The first three: negative, neutral, and positive—are normalized to the range from 0 to 1 and add up to unity. The scores of *eggy_the_elf*’s epistle are 0.363, 0.572, and 0.065, confirming our fears that the message feels more negative than positive. The fourth, compound score ranges from -1 to +1. It is calculated separately (not directly related to the first three scores) and represents the sentiment level directly (a very negative -0.9494 in the case of *eggy_the_elf*).

As a pre-trained regressor, VADER works well on average but can be improved to deal with domain-specific texts (such as texts on Politics and Economics). For example, the word “inflation” has a negative connotation in Economics but is relatively neutral in the Physics context and even positive in the car tire industry. You can train a customized Naïve Bayesian classifier (see [Chapter 6](#) on p. 101) or any other classifier, for that matter, to calculate customized sentiment level. To train the new classifier, you need a set of texts with known sentiments spanning a significant range⁷. Using the presence/absence of the constituent words/lemma as independent variables and the sentiment level as the dependent variable, you can follow the standard machine learning path ([Subsection 6.4.2](#) on p. 116) of training and testing and get a customized sentiment analyzer at the output.

⁷Classification is not limited to the “bad”–“good” scale. It could be “long”–“short,” “sweet”–“bitter,” or pretty much any other bipolar scale.

8.6 Topic Discovery

Topic discovery, or topic modeling, is an unsupervised machine-learning approach identifying topics within a text corpus. These topics are clusters of words/lemmas that share similar themes and tend to appear together. The criterion for co-occurrence—whether words are in the same sentence, paragraph, or within a specified proximity—varies. Generally, the broader the co-occurrence scope, the less coherent the identified topics may become.

You can use topic discovery to categorize large sets of unstructured text, making them more manageable and understandable. Topic discovery may lead to uncovering latent themes not apparent through manual reading, providing deeper insights into the content. Topic lemmas can serve as keywords in search and information retrieval processes. Last but not least, computed repeatedly, topic discovery can identify emerging trends and sentiments.

The most popular method for topic discovery is Latent Dirichlet Allocation (LDA). Proposed in 2000–03, LDA is a statistical model that identifies topics in a collection of documents. LDA assumes that each document is a mixture of topics and each topic is a mixture (a “bag”) of words or lemmas. LDA iteratively associates words with topics and topics with documents. These associations are further used to group similar documents together and similar words into topics. The output is a set of topics characterized by a cluster of words, with each document being a mix of these topics to varying degrees.

For example, consider the topic discovery in “Alice’s Adventures in Wonderland” by Lewis Carroll. The full text of the fantasy tale can be downloaded from the Project Gutenberg website⁸. It is also available as a part of the Gutenberg Corpus in the Natural Language Toolkit (NLTK). We instruct the topic modeler to produce five topics. Since “Alice” is one document, but we need many documents to apply LDA, we split it into 828 paragraphs and treat each as a separate document.

The results of topic discovery are strongly influenced by frequently used words, which also happen to be either stop words (removed elsewhere in the pipeline) or references to the protagonist(s). Indeed, a book about Alice would mention Alice in almost every paragraph, making the little girl’s name the most frequently used member of each discovered topic. Consider such names as stop words, too.

The modeler’s full output comprises a table with five columns (one per topic) and 1,636 rows (one per lemma). For clarity, the modeler also reports each topic’s top ten most frequently used words and their frequencies, as shown in Table 8.3.

Note that several lemmas, such as “little,” “time,” “large,” “King,” “Queen,” “Turtle,” and “Hatter,” dominate in each topic, making the differences between the topics subtle and their interpretation challenging. However,

⁸<https://www.gutenberg.org/ebooks/28885>

TABLE 8.3
Five topics in “Alice’s Adventures in Wonderland” by Lewis Carroll and their most frequently used words with frequencies in 1/1000th.

T1	T2	T3	T4	T5
little 21	little 13	time 16	Cat 17	little 15
large 11	turn 10	King 12	Queen 13	Turtle 15
Hatter 8	way 10	little 11	Dormouse 13	Mock 14
Mouse 7	Gryphon 9	Duchess 10	Hatter 12	Queen 11
grow 7	Hatter 9	course 7	way 9	voice 10
way 7	Turtle 8	Rabbit 7	King 8	tone 9
door 7	sit 7	moment 6	nothing 8	Caterpillar 8
time 7	King 7	mean 6	Mouse 8	time 7
head 7	head 7	Caterpillar 6	little 8	last 7
March 7	minute 7	turn 6	Gryphon 8	King 7

if you are at least remotely familiar with the plot, you may assign meaning to the topics or seek help from an LLM or another abstractive summarizer:

Variability and Exploration: Focuses on themes of size change, interactions with characters like the Hatter and the Mouse, and navigating through environments like doors.

Adventurous Encounters: Highlights Alice’s dynamic interactions with characters such as the Gryphon, Hatter, and Turtle, suggesting moments of decision and contemplation.

Time and Social Dynamics: Emphasizes the concept of time and social interactions with characters like the King and Duchess, reflecting on the complexity of communication and social norms.

Character Interactions and Perspectives: Centers around Alice’s interactions with characters like the Cat, Dormouse, and Queen, indicating varied perspectives and conversations.

Emotional Expressions and Dialogues: Showcases emotional dialogues involving characters like the Mock Turtle, highlighting tones and moods in conversations.

An important but often overlooked aspect of topic discovery related to [Chapter 5](#) is the nature of the relationships it reveals. Words grouped into the same topic demonstrate a statistical tendency to appear together in texts. This tendency represents a correlation, not necessarily a direct or causal link between the words. Understanding this distinction is crucial for correctly interpreting the results of topic discovery.

8.7 Discussion Questions

1. How does denoising impact the accuracy and efficiency of NLP tasks? Discuss the significance of removing or replacing HTML tags and entities before text processing.
2. Considering the challenges of language detection described, debate the effectiveness of character n-gram analysis versus dictionary-based methods in accurately identifying the language of a document.
3. Evaluate the roles of stemming and lemmatization within the Text Processing Pipeline. How do they differ in approach, and what are the implications of choosing one over the other for specific NLP applications?
4. Compare the summaries of this chapter on pp. 156 (extractive) and 158 (abstractive). Look into their clarity, representativeness, and comprehensiveness.
5. Reflect on the implications of named entity recognition (NER) inaccuracies for social network analysis and sentiment analysis. How do the challenges of NER, such as misrecognition or misclassification, affect the outcomes of these analyses?



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Part II

“Technical Track”



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

Introduction

Technical Track aims to equip you with a profound grasp of the Python programming language, and the computational prowess needed in quantitative social science. While it may seem like another Python textbook at first glance, it is far from it.

Learning a comprehensive programming language like Python for the sake of mastering the language is no easy task. The language comprises many complex components that, when skillfully orchestrated, can solve many problems—from creating a computer game to performing stock market analysis and even controlling a Mars rover. An all-encompassing Python book tries to cover most, if not all, of these components since it does not assume any particular area of expertise from you. Interestingly, a traditional book might focus on the language’s basics, asserting that, if intrigued, you can independently explore any specific topic of interest.

However, the authors of this book have the advantage of knowing—or at least making an educated guess—about your profile. The book, particularly the Technical Track, is designed for qualitative social science researchers. This insight allows us to tailor the content to the challenges a social scientist will likely tackle using Python.

The track’s opening chapter may resemble a standard Python tutorial, albeit with examples alluding to social science. Regrettably, one cannot do much without understanding the powerful conditional statements, loops, and third-party modules discussed in the subsequent chapters.

Conversely, cramming all the language’s foundational principles into one chapter would render it dry, difficult to comprehend, irrelevant, and irksome. Our approach is to provide you with a “first-aid programming kit” to familiarize yourself with Python and pique your colleagues’ curiosity. As you progress through the track, you will continually expand your kit by gathering the necessary tools for data acquisition and exploration, probability and causality, hypothesis testing, predictive analytics, complex network analysis, text analysis, and data classification.

Every chapter concludes with several programming exercises. It is crucial to practice Python by replicating examples from the text and programming independently. Like learning a foreign language, you master programming by actively engaging in it—or, in Python’s case, writing code. Some exercises include solutions ([Chapter A](#)). Try to solve the exercises on your own before referring to these solutions.

On Notation

The following notation is used throughout the Technical Track. In the text, any Python expressions, including string and numeric constants, are shown in monotype font and enclosed in backticks: ``print("Hello, world!")``. The examples are in monotype font, too, but Python keywords and built-in functions are also emphasized in bold: ``print("Hello, world!")``. Program console output is prefixed with ``#>``:

```
1 #> Hello , world!
```

Line numbers are added for convenience on the left when needed.

Discussion Question

1. How does integrating Python programming skills benefit quantitative social science research? Provide examples of how such skills might have significantly contributed to breakthroughs in this field.

Python Programming Fundamentals

As a quantitative (or wannabe quantitative) social scientist turning into a programmer (“coder”), you probably know how to work with Excel or a similar spreadsheet, such as Google Sheets or OpenOffice/LibreOffice¹. This Chapter assumes that you are reasonably comfortable with spreadsheets. It also assumes that you do not know much about computer programming yet. If you do and start feeling bored, you can skip some of the topics in this Chapter. Use your best judgment.

9.1 Programming Workflow

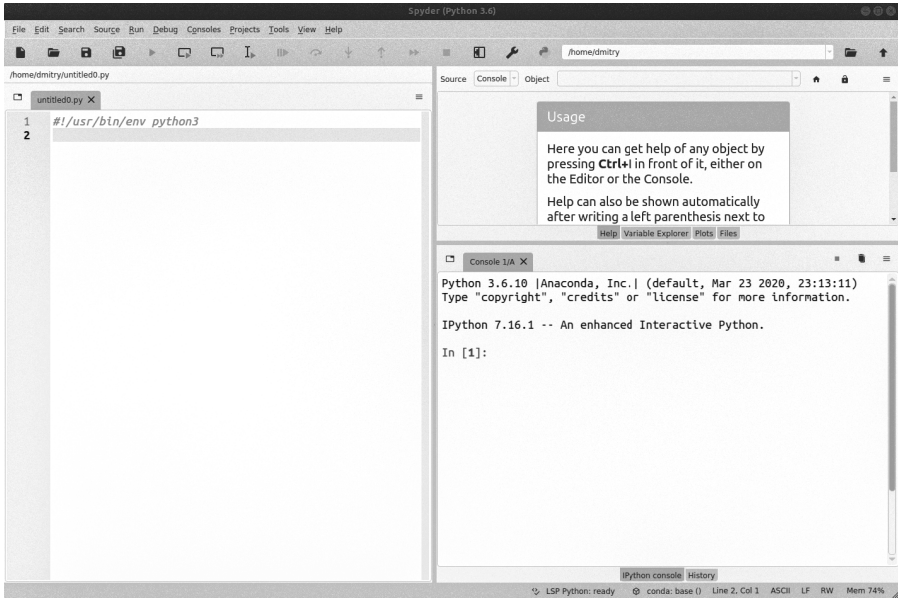
The ultimate goal of a computer program is to solve your problem or problems. Any non-trivial problem can be solved in many ways. Any solution must be at least correct but also efficient (fast and having a small memory footprint) and maintainable (easy to understand and modify). The three features: correctness, efficiency, and maintainability—often compete, turning your solution into a nasty compromise. To this extent, coding is a wicked problem, not unlike creative writing.

Your job as a coder is translating your problem from a language you understand (a natural language, such as English) into code: a language your computer understands (a programming language, such as Python). Assuming you know English, this Chapter will teach you Python—a programming language—and basic translation skills.

Writing a program starts with a deep understanding of the problem: its inputs, processing procedures, outputs, and performance considerations. In the later Chapters, you will explore more profound program design principles for computational social science. Now, you will experiment with smaller (“toy”) problems and programs.

Ready to start coding yet? Start your favorite integrated development environment (IDE).

¹We will refer to Excel and all other spreadsheets as “spreadsheets” for brevity.

**FIGURE 9.1**

Spyder IDE with an empty new file in the editor window and Python 3.6.10 prompt in the console window.

9.1.1 Integrated Development Environments

Your first step is to download and install a Python distribution: a collection of software development tools that includes Python itself, the Python standard library, an integrated development environment, and (optionally) third-party components. In this book, we assume that you use Anaconda². At the time of writing the book, Anaconda is free for non-commercial developers and will hopefully stay free forever.

An integrated development environment (IDE) is a computer program that simplifies software development by combining the capabilities for editing, executing, debugging, and testing your code. Specialized IDEs also support database connectivity, data visualization, and other goodies. One can indeed code without an IDE, but using an IDE makes this already painful process less painful, especially for beginners.

Figure 9.1 shows a screenshot of Spyder, a free and popular Python IDE, a part of Anaconda. An IDE is simply a fancy environment around a Python interpreter. The correctness of your code does not depend on the choice of an IDE.

The upper right Spyder's window contains various support tools. The Python console is in the bottom right corner. It is used for interaction with

²<https://www.anaconda.com/>

Python: typing individual commands and small code fragments and observing their behavior. The editor window is on the left. It is used to write complete programs. A coder often experiments with code on the console command line and later copies suitable fragments into the program file.

Traditionally, the first program one writes in a new programming language displays “Hello, world!” on the console. Type the following code on the console precisely as shown. Take special care of the quotation marks: they could be single or double, but matching and straight (not fancy). If you come from C/C++/Java programming, teach yourself not to end a command with a semicolon.

```
print("Hello, world!")
```

Press **Enter** and enjoy the output of your first Python program. Note that neither the command `print()` nor the quotation marks are displayed.

```
#> Hello, world!
```

Let’s try it again with single quotation marks:

```
print('Hello, world!')  
#> Hello, world!
```

Python does not distinguish between matching single and double quotation marks, nor should you. However, an attempt to cheat causes an immediate burst of outrage, also known as an error message.

```
print("Hello, world!")
```

Does this look intimidating?

```
#> File "<stdin>", line 1  
#>     print("Hello, world!")  
#>                                     ^  
#> SyntaxError: EOL while scanning string literal
```

Despite the look, error messages are a programmer’s best friends: they inform you about the type of the error (`SyntaxError. . .`), the file and line number of the command that caused the error (`File "<stdin>", line 1`), and the command that caused it. You will read more about errors in [Section 11.2](#) on p. 219.

9.1.2 Program Execution

When you press **Enter** on the console command line, the command is sent to the Python interpreter. An interpreter is another program (conveniently called “python”) that executes the code in a programming language. Using the above example, the interpreter recognizes the word `print` (a command to

display something on the console), followed by a quoted message in parentheses ``("Hello, world!")``. Such commands are called functions, and the content in the parentheses are the function arguments.

The command is known to Python and is executed by displaying the message. The interpreter then proceeds to read more of your commands. This behavior is called Read–Execute–Print–Loop (REPL).

When you code a program in a separate file, the interpreter reads the commands from the file instead of the command line but executes them similarly. If you plan to reuse your code in the future, you must save it into a file with an extension `.py``—for example, `hello.py``:

Listing 9.1

hello.py

```
1 # Written by DZ
2 print("Hello, world!")
```

Consider choosing meaningful names for the files. You will need them to find your programs later.

A part of a line in a Python file that starts with a pound sign ``#`` is called a comment. Python ignores comments. They are meant for you and other future readers of your code. Comments document your design choices and provide additional context for the commands. Do not be shy of documenting your code: You will not remember why you wrote it this way or that two weeks after you wrote it.

9.2 Variables, Data Types, Operations, and Operators

A program generally manipulates previously stored text, numeric, and mixed data. This section introduces data storage and data manipulation mechanisms.

9.2.1 Variables

When you work with a spreadsheet, you paste your data into cells. One cell can hold a number (integer or real, also incorrectly known as “decimal”) or a text fragment. You can refer to the content of a cell by the column name and row number. For example, ``=H12`` is the content of the cell in column ``H`` and the 12th row. Here, the equal sign ``=`` helps the spreadsheet to distinguish text fragment ``H12`` (wrong) and a reference to a cell (right). If the cells ``H12`` and ``B3`` contain numbers 3.0 and 0.14159, respectively, then the formula ``=H12+B3`` calculates 3.14159, the sum of them.

A Python equivalent of a cell is called a variable. A variable stores a data item, which is the value of that variable. It has a human-readable name (not a

grid-inspired name). A variable name is case-sensitive. It begins with a letter or an underscore and contains only letters, digits, and underscores. A variable name reflects the purpose of the variable:

- The variable ``year`` presumably contains a year.
- The value of the variable ``GDP`` is a country's gross domestic product.
- One would expect to find the text of the U.S. Constitution in the variable ``us_constitution``.

```
1 year = 2022 # "year" is created and becomes 2022
2 year = year + 1 # now, "year" becomes 2023
3 GDP = 20.94 # "GDP" is created and becomes 20.94
4 us_const = "We the People of the United States..."
5 del year # "year" is no more
```

The first line of the code fragment creates a variable ``year`` or changes its value if the variable already exists. Such an operation is called “assignment.” It looks like an equal sign `=`, but it means “take the value on the right and store it in the variable on the left.” In other words, the value of the variable *becomes* 2022. An assignment operation is one of many statements available in Python. You will see other statements later.

Line 2 is the most intriguing: it accesses the old value of the ``year`` (2022), adds 1 to it, and stores it under the same name. The old value of the variable is gone. You need two variables to have both the old and the new values.

Finally, the ``del`` statement deletes a variable that is unnecessary anymore—but do not worry, this is a rare operation. Python usually takes good care of unwanted items.

9.2.2 Numbers

Python recognizes two types of numbers: integer numbers (without a fractional part) and real numbers, also known as floating-point numbers. Integer numbers are precise. Use them for counting: 10 fingers, 435 representatives, 250 years. Floating-point numbers are approximate. Use them to measure: 6.0 feet, 99.7 kilograms, and 4.5 million residents. The functions ``float()`` and ``int()`` convert any number to a floating-point or integer number. The latter function does not round but truncates the fractional part:

```
1 f_year = float(year)
2 i_GDP = int(GDP)
3 print(year, f_year, GDP, i_GDP)
4 #> 2023 2023.0 20.94 20
```

All numbers can be added, subtracted, multiplied, and divided using the same operations as in a spreadsheet. The respective symbols ``+``, ``-``, ``*``, and

`/` are called operators, and the numbers involved in the operation are called operands. Operator `/` performs integer division (the result is truncated to an integer number). Operator `%` calculates the remainder. Operator `\*\*` calculates the power. If any operand is a floating-point number, the result of the operation is also a floating-point number:

```
print(1+3.1, 55*(1-0.7), 49/10, 49//10, 49%10, 4**5)
#> 4.1 16.5000000000000004 4.9 4 9 1024
```

Note that, according to Python, `55\*(1-0.7)` is 16.5000000000000004, not 16.5, as one would expect. The result is incorrect because floating-point numbers are approximate. Another observation: the function `print()` can display multiple results. Separate them with commas.

If a number is stored in a variable because it was previously calculated, you can (and should) refer to that variable instead of retyping the number. Variables make it possible to reuse the previously computed values. That was and is their original purpose. Meaningful variable names make your code easy to read and understand:

```
1 population = 332_403_650 # _ is a thousands separator
2 growth_rate = 0.004 # 0.4%
3 population = population * (1 + growth_rate)
4 print(population)
5 #> 333733264.6
```

But what about text fragments and other symbolic data, such as names and addresses? For them, Python has strings.

9.2.3 Strings

A string is a sequence of printable symbols (characters) enclosed into single or double quotation marks. `"Hello, world!"` is a string, and the value of the previously defined variable, `us_const`, is a string, too. Strings are mainly intended for human-computer interaction (read commands and display messages) and natural language processing.

Strings can contain special two-symbol combinations known as special characters. The first symbol is always a backslash, an escape character. Here is a brief list of the special characters:

- The special character ``\\n`` represents a line break. It still counts as one character!
- ``\\t`` stands for a tab space.
- ``\\\"`` is a double quote within a double-quoted string.
- ``\\'`` is a single quote within a single-quoted string.

- `\\` is a backslash itself.

Function `len()`, the “universal yardstick,” calculates the length of a string: the number of characters in the string, including spaces and special characters.

```
print(len("Hello ,_\\"world\\"!\n"), len(us_const))
#> 16 37
```

Strings are immutable: once created, they cannot be changed but can be copied with modifications, split into fragments, and combined. Let’s have a look at some string operations.

Functions `int(x)`, `float(x)`, and `str(x)` convert `x` into an integer number, floating-point number, and string, if possible, regardless of the nature of `x`. Operators `+` and `*` combine (“concatenate”) two strings and create multiple copies of a string:

```
1 country = "USA"
2 first_name = "George"
3 last_name = "Washington"
4 full_name = first_name + "_" + last_name
5 print(2 * country + "_" + full_name + "_" + 2 * country)
6 #> USAUSA George Washington USAUSA
```

Note that these operators behave very differently when applied to strings and numbers. The following example vividly illustrates the difference:

```
print(123+321, "123"+"321", int("123")+float("321"))
#> 444 123321 444.0
```

If your “number” is represented as a string (e.g., `"1776"`), you must convert it to an actual number before further operations.

You can extract an individual character from a string using the indexing operator: a pair of square brackets `[]` (not to be confused with curly braces `{}` and parentheses `()`). The operator needs to know the index (position) of the character to extract. Historically, the first character of a string is at position number 0; the second is at position number 1, *etc.* Also, the last character is at position number -1; the second from the end is at position -2, *etc.* You can use positive or negative indexes, whatever is more convenient.

```
# "We the People of the United States..."
print(us_const[0], us_const[1],
            us_const[-1], us_const[-4])
#> W e . s
```

It is time to solve a small but not entirely useless problem: Display the middle letter of an unknown string. If the string has an odd length, the central character’s index is half the length (use integer division by 2 to obtain an

integer index). If the length is even, there are two “middle” characters, neither indeed in the middle. You can suppose that the one on the right of the true middle is the true middle character. Its index is, again, half the length of the string.

```
city = "Boston"
middle = len(city) // 2
print(middle, city[middle])
#> 3 t
```

You can extract a range of consecutive characters from a string using the slicing operator. The operator looks like a pair of square brackets but has up to three arguments separated by colons `:`. If present, the first argument designates the beginning of the slice: the index of the first included character. If it is absent, the slice begins at the beginning of the string. The second argument, if present, designates the end of the slice: the index of the first character that is not included. If it is absent, the slice ends at the end of the string. The third argument defines the step. Each character in the range is extracted if the step equals 1 or is absent. If the step equals 2, every other character is extracted, *etc.* If the step is negative, the extracted slice is reversed (spelled backward). You do not need the second colon if the step is not used. Here are code fragments with examples galore.

- Print the city name from the beginning (no index before the colon) to the end (no index after the colon):

```
print(city[:])
#> Boston
```

- Print the city name from the beginning to the character number 3—in other words, the first three characters:

```
print(city[:3])
#> Bos
```

- Print the city name from character number 3 to the end—in other words, everything except the first three characters:

```
print(city[3:])
#> ton
```

- Print the city name from character number 2 to the second character from the end—in other words, everything except the first two and the last two characters:

```
print(city[2:-2])
#> st
```

- Print every other character of the city name:

```
print(city[::2])
#> Bso
```

- Print every other character of the city name, starting from 1:

```
print(city[1::2])
#> otn
```

- Print the city name backward, with a negative step:

```
print(city[::-1])
#> notsoB
```

The most potent and helpful string operations are available through so-called string methods: functions that are attached to strings³. There are nearly 45 of them, but you need just half for starters. As a side note, the source of Python’s power is in its functions and modules ([Section 9.3](#) on p. 185). The more of them you learn to use, the fewer wheels you will have to reinvent.

String functions in the first group check strings’ properties. They produce (“return”) values ``True`` and ``False``, which are neither strings nor numbers. ``True`` and ``False`` are boolean values, named after George Bool, one of the founders of formal logic. You will use them later to check conditions and steer program execution ([Section 11.1](#) on p. 217). Functions that return boolean values are often called predicate functions and answer yes-or-no questions ([Table 9.1](#)).

```
print(city.isupper(), city.isalpha(), "2023".isdigit(),
      "____".isspace(), city.endswith("don"))
#> False True True True False
```

[Table 9.2](#) shows string conversion functions. Since Python strings are immutable, these functions do not change the originals: They produce their converted copies. Bear this in mind when you work with large texts: Each step in a seemingly innocent sequence of conversions produces a copy of the size of the original.

```
print(city.upper(), city.swapcase(), "_2023_".strip(),
      city.replace("ton", "tin").replace("o", "ah"))
#> BOSTON bOSTON 2023 Bahstin
```

³For the rest of the book, we will call methods “functions,” as long as it does not cause any misunderstanding.

TABLE 9.1

Predicate string functions

<code>`str.isalnum()`</code>	Are all characters in the string alphanumeric?
<code>`str.isalpha()`</code>	Are all characters in the string alphabetic?
<code>`str.isdigit()`</code>	Are all characters in the string decimal digits?
<code>`str.islower()`</code>	Are all characters in the string in lower case?
<code>`str.isspace()`</code>	Are all characters in the string whitespaces?
<code>`str.isupper()`</code>	Are all characters in the string in upper case?
<code>`str.endswith()`</code>	Does the string end with the specified value?
<code>`str.startswith()`</code>	Does the string start with the specified value?

TABLE 9.2

Conversion string functions

<code>`str.capitalize()`</code>	Converts the first character to upper case
<code>`str.lower()`</code>	Converts a string into lower case
<code>`str.lstrip()`</code>	Trims the string (remove spaces) on the left
<code>`str.replace(x,y)`</code>	Replaces substring <code>`x`</code> with substring <code>`y`</code>
<code>`str.rstrip()`</code>	Trims the string (remove spaces) on the right
<code>`str.strip()`</code>	Trims the string on both ends (but not inside)
<code>`str.swapcase()`</code>	Swaps the characters' cases
<code>`str.title()`</code>	Converts first character of each word to uppercase
<code>`str.upper()`</code>	Converts a string to upper case

The last expression in the example shows how string functions can be chained. In ``city.replace("ton", "tin").replace("o", "ah")``, the first function replaces ``"ton"`` with ``"tin"`` and returns a new string ``"Bostin"``. The new string is "anonymous"; it is not stored in any variable but is immediately processed by the next ``str.replace()``.

Table 9.3 shows string search functions. The difference between otherwise nearly identical functions ``str.find(x)`` and ``str.index(x)`` is how they handle a failure. If ``x`` cannot be found in ``S``, then ``str.find(x)`` returns a -1, but ``str.index(x)`` displays an error message. Remember that the first character of a string has an index of 0, not 1.

The expression ``x in S`` last row is not a string function. It checks whether some substring is part of a (larger) string. You will see it again later: generally, it checks whether any item ``x`` is a part of any collection ``S``.

TABLE 9.3

Search string functions

<code>`str.count(x)`</code>	Counts the number of occurrences of <code>`x`</code>
<code>`str.find(x)`</code>	Searches for a substring <code>`x`</code> and reports its position
<code>`str.index(x)`</code>	Searches for a substring <code>`x`</code> and reports its position
<code>`x in S`</code>	Is substring <code>`x`</code> a part of <code>`S`</code> ?

```

print(city.count('o'), city.find('o'), city.index('o'),
      city.find('a'), "a" in city)
#> 2 1 1 -1 False
print(city.index('a'))
#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> ValueError: substring not found

```

Two more string functions, `str.split()` and `str.join()`, will be introduced in [Subsection 10.1.1](#) on p. 189 because using them requires knowledge of lists.

While you are still not in a position to solve practical text analysis problems in Python, you can do some elegant exercises, such as drawing a frame of stars around an arbitrary string.

1. First, let's define the name of the city (or any other message you want to frame) and the symbol used for framing as variables `text` and `style`.

```

text = "Boston"
style = "*"

```

2. Next, calculate the width of the frame. It equals the number of characters in the text `len(text)`, plus the width of the margins on both sides (two more characters), plus the width of the frame on both sides (two more characters).

```

width = len(text) + 2 + 2

```

3. Proceed to the construction of the horizontal lines of stars at the top and bottom of the frame: take the frame character, copy it as needed, and add a line break at the end.

```

hline = style * width + "\n"

```

4. Create a spacer line that will be placed above and below the text. The spacer starts with a star symbol, followed by `width-2` spaces to cover the text and the margins, and ends with another star and a line break.

```

spacer = style + " " * (width-2) + style + "\n"

```

5. The last line to construct is the line with the text. It starts with a star, followed by a space, followed by the text itself, followed by another space, and ends with a star and a line break.

```

text = style + " " + text + " " + style + "\n"

```


6. You are almost done! Compose the frame line by line and display it.

```
frame = hline + spacer + text + spacer + hline
print(frame, end="")
```

You can find the complete code of the example in the following listing (“listing” is how programmers call their code files with line numbers for easy referencing):

Listing 9.2

frame.py

```
1  # Define
2  text = "Boston"
3  style = "*"
4
5  # Construct
6  width = len(text) + 4
7  hline = style * width + "\n"
8  spacer = style + " " * (width-2) + style + "\n"
9  text = style + " " + text + " " + style + "\n"
10 frame = hline + spacer + text + spacer + hline
11
12 # Display
13 print(frame, end="")
14
15 #> *****
16 #> *           *
17 #> *  Boston  *
18 #> *           *
19 #> *****
```

Things to note:

- You can control the text and the frame style by modifying the namesake variables.
- By editing the program code, you can control the spacing on the sides, above, and below the text. Not an ideal solution, indeed.
- A line break at the end of the frame is inconveniently introduced as a part of the `hline` variable. You must tell function `print()` to suppress its additional line break by supplying an optional argument `end=""`. By default (if not specified), `end="\n"`. A pair of quotation marks `""` is an empty string with no characters—which is not the only oddity in Python.

TABLE 9.4
Truth table for the logical operators

`a`	`b`	`not a`	`a and b`	`a or b`
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

9.2.4 Data Types

You may still have some doubts about the difference between strings and numbers. Strings and the two flavors of numbers are three different data types. A data type is a collection of items that possess the same properties and support (or are subject to) the same operations. For example, all integer numbers can be used for counting (yes, even the negative numbers when one is in the red); they can be added, subtracted, divided, *etc.* All numbers can be compared with the relational operators `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), `==` (equal), and `!=` (not equal).

All strings can contain one or more characters (they are containers); they can be concatenated, searched, converted, or “multiplied” (replicated).

The boolean values `True` and `False` are a data type of their own. They do not support any string or arithmetic operations but can be negated with the logical operator `not` and combined with the logical operators `and` and `or`, as prescribed by Table 9.4.

The table is known as a truth table. The first two columns contain every possible combination of two boolean variables, `a` and `b`, either of which can be true or false. Operator `not` in the third column negates the variable `a`: it converts a boolean condition into its opposite. Operator `and` in the fourth column checks if **both** conditions are simultaneously true. Operator `or` in the fifth column checks if **either** condition is true.

The meaning of `and` in Python differs from that of the “and” conjunction in English. In English, one can say, “She strolls on Mondays and Thursdays.” In Python, one must say `(day=="Monday") or (day=="Thursday")` because she cannot stroll on two days simultaneously. The condition `(day=="Monday") and (day=="Thursday")` is always false.

Another common mistake is using `<` when comparing a variable to more than one value, as in the following example:

```
day = "Monday"
weekend = bool(day == "Saturday" or "Sunday")
print(weekend)
#> True — ???
```

The operator `==` is computed before `or` (it has higher precedence). Essentially, the second line of the listing means:

```
weekend = bool((day == "Saturday") or "Sunday")
```

Since the first condition is false, the `or` operator proceeds to the second condition. The second condition is the string `"Sunday"` alone. Python treats any non-empty string as `True`, which is how Monday becomes a weekend day. Sadly, the right way to compare if a variable equals one of the values on a list is to check it against each value separately or use the `in` operator ([Subsection 11.1.1](#) on p. 218).

```
day = "Monday"
weekend = bool(day == "Saturday" or day == "Sunday")
print(weekend)
#> False
```

Here is another snippet that illustrates the usage of logical operators.

```
1 city_A_pop = 652_000.0
2 city_B_pop = 12_800_000.0
3 town_pop = 36_000.0
4 print(city_A_pop < town_pop)
5 #> False
6 print(city_B_pop != city_A_pop)
7 #> True
8 print(city_A_pop >= city_B_pop and city_A_pop > town_pop)
9 #> False
10 print(city_A_pop >= city_B_pop or city_A_pop > town_pop)
11 #> True
```

A value `None` is the only member of the `NoneType`. It is used as a placeholder for cases when some value is required, but its specific type is not known or is not essential.

Python natively supports more than a dozen data types. In addition to the already familiar `NoneType`, `bool`, `int`, `float`, and `str`, it has data types for lists (`list`), dictionaries (`dict`), sets (`set`), tuples, or immutable lists (`tuple`), and more. Some of them will be reviewed in [Subsection 10.1.1](#) on p. 189.

Python modules often provide additional data types. For example, module `numpy` adds data type `ndarray`, and module `pandas` ([Section 10.3](#) on p. 197) adds dataframe and series. New data types introduce new properties and new operations. They make programming faster and more efficient, but you must invest time and effort in learning about them.

To conclude with data types (so far), function `type(x)` reports the type of object `x`. When in doubt, use this function to discover the nature of the object.

```

1 city = "Boston"
2 population = 689_326.0
3 print(type(city), type(population), type(1 > 2))
4 #> <class 'str'> <class 'float'> <class 'bool'>
5 print(type(city)==str, type(2023)==int)
6 #> True True

```

9.2.5 Reading and Displaying Data

Most social science programming projects are “big data” projects based on relatively large datasets. It is doubtful that you or your program users will manually enter data on the console. More likely, the data will be read from files (spreadsheets, text files, XML files; [Chapter 10](#) on p. 188) or pulled from databases. It does not mean your software will never use the console for input: you may want to enter commands, options, arguments, or other fragmentary information. Function `input()` is the Python tool for reading data on the console.

In its vanilla form, the function requires no arguments. It silently stops the program and awaits the user’s input. When the user types anything followed by the **Enter** key, the function collects the input and returns it to the program as a single string. Most commonly, the string is assigned to a variable for further processing. If your code expects more than one input, using the prompt (an invitation to enter) as the function’s argument is recommended. The prompt will be displayed on the console just before stopping the program to inform the user about the expected input.

```

1 city = input("Enter the city name: ")
2 print("You entered \"" + city + "\".")
3 #> Enter the city name: Boston
4 #> You entered "Boston".

```

Suppose the input is expected to be numerical (integer or float). In that case, immediately converting the input string to the required data type is a common practice.

```

1 n1 = float(input("Enter the first number: "))
2 n2 = float(input("Enter the second number: "))
3 print("The sum is " + str(n1 + n2))
4 #> Enter the first number: 2.44
5 #> Enter the second number: 3.17
6 #> The sum is 5.609999999999999

```

Things to note:

- The strings returned by the `input()` functions are not assigned to variables but immediately and anonymously passed as arguments to the type converters `float()`.

- The addition result is converted back to a string before the concatenation because strings cannot be concatenated with numbers.
- The addition result is incorrect because floating-point numbers are approximate (p. 174).

Another common input transformation is normalization: Removing white spaces at both ends of the entered text and converting the stripped text to a “standard” character case (upper or lower) for further case-insensitive analysis.

Listing 9.3

Text normalization

```

1 city = input("Enter the city name: ").strip().upper()
2 print("You entered \" " + city + "\".")
3 #> Enter the city name:  Boston
4 #> You entered "BOSTON".

```

Once again, like on page 178, the string produced by `input()` becomes the base for the string function `str.strip()`, and the output of the latter becomes the base for `str.upper()`.

The output of a data analysis program is usually much less copious than the input. You are still encouraged to save the results to a file. Still, it does not preclude you from displaying some of them on the console for immediate evaluation. Incidentally, you are already familiar with the function `print()`, which does just that. The function can print a list of comma-separated arguments, in which each comma is replaced in the printout with a single space (line 2 of Listing 9.3). However, if you prepare one formatted string to display, you will have more control over the printout.

A formatted string has the prefix `f` (lowercase “f”) immediately before the opening quotation mark. It contains special inclusion characters: one or more pairs of matching curly braces `{}`. A part of a string within a pair of braces is treated as valid Python code. It is executed and replaced with the execution result.

```

1 city = "Boston"
2 population = 689_326.0
3 message = f"{city.upper()}: {int(population)} residents."
4 print(message)
5 #> BOSTON: 689326 residents.

```

Formatted strings support many advanced functions, such as controlling the number of decimal digits after the decimal point, padding numbers with zeros in front, and padding strings with spaces on either or both sides for tabular output. These functions are outside of this book’s scope.

TABLE 9.5
Notable modules in the Python standard library

Module	Brief description
<code>`calendar`</code>	Calendar-related functions (e.g., look up day by date)
<code>`collections`</code>	Useful data types (e.g., <code>`Counter`</code>)
<code>`csv`</code>	Support for comma-separated values (CSV) files
<code>`datetime`</code>	Date and time-related functions
<code>`itertools`</code>	Handy functions to work with data sequences
<code>`json`</code>	Support for JSON files
<code>`math`</code>	Mathematical functions (e.g., $\sqrt{x}$ and $\sin(x)$)
<code>`pickle`</code>	Functions for saving and restoring the project data
<code>`random`</code>	Random number generators
<code>`re`</code>	Tools for searching text with patterns
<code>`xml`</code>	Support for XML files

9.3 Modules

The power of Python lies not only in its data types, operators, and functions but also in the Python standard library: a collection of nearly 250 “toolboxes” of functions and data types called packages or modules. Each module has a unique name and is stored in one Python file. The content of a module is unavailable to the programmer unless explicitly imported. Table 9.5 lists some notable modules in the Python standard library. You will get familiar with them elsewhere in the book.

As a rule, all standard modules are installed together with Python. To use a standard module, import it using the ``import`` command. Refer to the functions and variables within the module by their “full name”: module name dot function/variable name.

```
1 import math
2 print(math.sin(math.pi))
3 #> 1.2246467991473532e-16
4 print(f"{math.sqrt(4)} is {4**0.5}")
5 #> 2.0 is 2.0
```

Note that $\sin(\pi) = 0$ mathematically, but floating point numbers are inaccurate. The closest Python could find to 0 was ``1.2246467991473532e-16``—a small but positive number. If you are not familiar with scientific notation, ``1.2246467991473532e-16`` is a way to represent minuscule numbers and is equivalent to $1.2246467991473532 \times 10^{-16}$. Scientific notation works for gigantic numbers, too: ``1e100`` is 1×10^{100} , a one followed by one hundred zeros, also known as a googol.

Another example involves modules ``random`` and ``math``.

```
1 import random, math
2 print(math.sin(random.random()))
3 #> 0.8261688251699373
4 print(math.sin(random.random()))
5 #> 0.024759338998374195
```

Whenever you call the function ``random.random()``, it returns a new uniformly distributed random number between 0 and 1. Random numbers are used in computer simulations, randomized experiments, and machine learning.

Technically, any Python file is a module, but a well-designed module is consistent (containing only relevant functions and variables) and well-documented. You will learn to develop modules and properly document your code later.

9.3.1 Installing Extra Modules

In addition to the standard library modules, Python developers who are Anaconda users have access to more than a thousand third-party modules (such as ``numpy`` and ``pandas``). If the module you need has not been installed, you can install it. Start the Anaconda Navigator, and open the “Environments” tab. The window on the right shows the list of modules currently installed and modules available for installation.

Some rarely used modules may not be included in Anaconda. You can still try to install a missing module by executing ``!pip install module_name`` (with an exclamation point) in the Spyder console. Test any newly installed module by trying to import it.

9.4 What’s Next

In fewer than two dozen pages, you progressed from a spreadsheet user to a beginner Python programmer. In the next Chapter, you will learn to work with advanced “container” data types—lists, dictionaries, tuples, and Pandas dataframes—and with formatted and unformatted text files.

9.5 Programming Exercises

Exercise 1. Reimplement code in the file ``frame.py``. Add two variables, ``h_margin`` and ``v_margin``, that specify the number of spaces before and after and blank lines above and below the framed name. Display the name in the proper frame.

Exercise 2. The population of Boston in 2020 was 689,326 residents. It has increased by 0.14% in 2021 and by 0.28% more in 2022. Create variables ``population``, ``rate2020``, and ``rate2021``, and write a code fragment that calculates and displays the population in 2022.

Exercise 3. Write a program that asks the user to enter a word, replaces all characters “o” (except for the last one) with an asterisk ``*``, and displays the result. Assume that the word contains at least one “o”. Hint: Run command ``help(str.replace)`` to learn how to limit the number of replacements.

Exercise 4. Write a program that obtains 10 uniformly distributed random numbers between 0 and 1 and displays the average of them.

10

Data Collection and Cleaning

The effectiveness of data analysis hinges on data quality ([Chapter 2](#) on p. 19). This relationship is often expressed with the phrase “garbage in, garbage out.” This Chapter will primarily concentrate on the “garbage in” part of the equation, explaining how to import data into Python. We will also briefly cover the “garbage out” component, addressing how to manage missing values. Moreover, we will explore additional core features of Python.

10.1 Containers

In the previous Chapter, you learned about simple data types: integer and floating-point numbers (representing quantitative data) and character strings (signifying qualitative data). However, a single number or string can describe only a limited range of entities.

Take, for example, a college student. A student is defined by numerous attributes: name (first, last, and potentially middle), gender, academic rank, and GPA, to name a few. These attributes include three string values (the names), a floating-point number (the GPA), a categorical variable (rank, which could be a string or a limited-range integer), and the gender (which could be a limited integer, a string, or a boolean, assuming the admissions office acknowledges only two genders). Thus, the attributes are heterogeneous.

All attributes referring to the same student should be grouped for clarity. It is not just a GPA—it is Alice’s GPA. It is not merely a gender—it is Bob’s gender.

Now consider a group of students taking a course together, forming a class. The class is homogeneous as all members share the same attributes—name, gender, academic rank, and GPA. The values of these attributes, of course, differ for each student. A class is somewhat like a string of characters, except that a student is not a character (a light-hearted pun intended). For instance, students in a class could be ordered alphabetically and identified by their position in the sequence. Subsets of students can be selected, classes can be concatenated, and even student duplication is theoretically possible (although it is unclear why one might want to do that).

However, students may prefer to be identified by labels rather than numbers. A combination of the first and last name can serve as a natural label, with the last name alone sufficing if it is unique within the class. Alternatively, school or government-issued ID numbers can provide anonymized labels. What is essential is that all labels are unique and can accurately identify students or other entities.

Organizing data collections requires containers, data types that can hold more than one data item. You will familiarize yourself with Python's lists, tuples, dictionaries, and Pandas' series and dataframe.

10.1.1 Lists and Tuples

A list in Python is a heterogeneous, linear data structure consisting of items of any data type in any sequence. Unlike strings, lists are mutable: they can be created, modified, and even destroyed. You can append new items, insert them at any position, or remove existing ones from the list. You can also reorder list items by sorting them.

List items can be accessed via integer indexes (p. 175), both positive (from 0 to ``N-1``, where ``N`` is the length of the list, i.e., the number of items in the list) and negative (from -1 to ``-N``). Like strings, lists can be compared, sliced, concatenated, searched, and “multiplied” (replicated). Function ``len()`` returns the length of a list. An empty list with zero items can be a placeholder for future data.

Here are some illustrative examples using simple lists. We'll begin by creating two lists, an empty and a populated, and then combine them:

```
students = [] # An empty list
students.append("Alice") # Add a new item
more_students = ["Bob", "Chuck"]
print(students + more_students) # Concatenate w/o modifying
#> ['Alice', 'Bob', 'Chuck']
students.extend(more_students) # Add a sublist
print(students, len(students))
#> ['Alice', 'Bob', 'Chuck'] 3
# Delete the list if it is too big
del more_students
```

Next, let's examine list access using indexes:

```
print(students[0]) # An item
#> Alice
print(students[1:]) # A slice
#> ['Bob', 'Chuck']
students[1] = "Brian" # Assignment
print(students[:])
#> ['Alice', 'Brian', 'Chuck']
del students[1] # Removing by position
```

```
students.insert(0,"Abby") # Add before the position
print(students)
#> ['Abby', 'Alice', 'Chuck']
```

Theoretically, you can start with any list (not necessarily empty) and transform it into another. In practice, you often read a list from a file ([Section 10.2](#) on p. 194), rearrange it, and eliminate unwanted items.

Lists can be rearranged by reversing and sorting using the corresponding functions, ``list.reverse()`` and ``list.sort()``. Both functions permanently modify the list in place. To make a copy of the original list, use the function ``list.copy()``. An alternative function, ``sorted(list)``, does not change the original list but returns a sorted copy.

```
students.reverse() # Returns None
print(students)
#> ['Chuck', 'Alice', 'Abby']
print(sorted(students))
#> ['Abby', 'Alice', 'Chuck']
students.sort() # Returns None
print(students)
#> ['Abby', 'Alice', 'Chuck']
```

Further modification operations include removing an item by value, regardless of its position. Let us eliminate Abby; she was not on the original list and is probably an imposter (officially, an outlier).

```
students.remove("Abby")
print(students)
#> ['Alice', 'Chuck']
students.remove("Abby")
#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> ValueError: list.remove(x): x not in list
```

The first removal operation works as expected, but the second fails as Abby is no longer on the list. Error handling will be explained further in [Section 11.2](#) on p. 219.

Tuples

A tuple can be thought of as an immutable list. It shares many attributes and operations with a list, except those that involve mutations such as appending, deleting, sorting, or reversing. Items in a tuple are enclosed in parentheses. Tuples are typically used to store collections of constants that are not expected to change. For instance, tuples can hold *stop words*—these are words like “the,” “a,” and “an,” that, while frequently used, do not carry significant meaning by themselves.

```
students = ('Abby', 'Alice', 'Chuck')
print(students, len(students), students[-1])
#> ('Abby', 'Alice', 'Chuck') 3 Chuck
del students[0]
#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> TypeError: 'tuple' object doesn't support item deletion
```

From Strings to Lists and Back

Splitting a text string into fragments, such as words, is a fundamental operation in Natural Language Processing (NLP), which we will discuss in more detail in [Chapter 16](#). At the heart of NLP is the function `str.split()`, which divides the string into a list of substrings, excluding spaces or other designated separators.

If you call the function without any arguments (which is generally recommended), it eliminates any contiguous combination of white spaces as defined by `str.isspace()` ([Table 9.1](#) on p. 178). The remaining items are then colated into a list. If an argument is provided, the function considers it as the separator.

In the upcoming example, note how the lengthy string produced by ChatGPT-4 is enclosed in triple quotation marks. They allow for line breaks within the str.

```
text = """NLP is at the core of the GPT-4 architecture,
as well as other versions of the GPT series developed by
OpenAI. NLP is a subfield of artificial intelligence that
focuses on the interaction between computers and humans
through natural language. The main objective of NLP is to
read, decipher, understand, and make sense of the human
language in a valuable way. It involves many tasks, such as
language understanding, language generation, translation,
and question answering."""
words = text.split()
print(words)
#> ['NLP', 'is', 'at', 'the', 'core', 'of', 'the', 'GPT-4',
#> 'architecture,', 'as', 'well', 'as', 'other',
#> 'versions', 'of', 'the', 'GPT', 'series', 'developed',
#> 'by', 'OpenAI.', 'NLP', 'is', 'a', 'subfield', 'of',
#> 'artificial', 'intelligence', 'that', 'focuses', 'on',
#> 'the', 'interaction', 'between', 'computers', 'and',
#> 'humans', 'through', 'natural', 'language.', 'The',
#> 'main', 'objective', 'of', 'NLP', 'is', 'to', 'read,',
#> 'decipher,', 'understand,', 'and', 'make', 'sense',
#> 'of', 'the', 'human', 'language', 'in', 'a', 'valuable',
#> 'way.', 'It', 'involves', 'many', 'tasks,', 'such',
#> 'as', 'language', 'understanding,', 'language',
```

```
#> 'generation', 'translation', 'and', 'question', 'answering. ']  
print(len(words))  
#> 75
```

The second example demonstrates an attempt to split the text into sentences using a period as a separator.

```
sents = text.split(".")  
print(sents)  
#> ['NLP is at the core of the GPT-4 architecture, \nas well  
#> as other versions of the GPT series developed by \n  
#> OpenAI', ' NLP is a subfield of artificial intelligence  
#> that \nfocuses on the interaction between computers and  
#> humans \nthrough natural language', ' The main objective  
#> of NLP is to \nread, decipher, understand, and make sense  
#> of the human \nlanguage in a valuable way', ' It involves  
#> many tasks, such as \nlanguage understanding, language  
#> generation, translation, \nand question answering', '']  
print(len(sents))  
#> 5
```

This attempt is successful because the text does not contain internal periods (for example, abbreviations) or sentences ending with exclamation points or question marks. If you tried to replace `". "` with `".?!"`, things would get more complicated. The function would try to use the entire string `".?!" ` as a separator, likely failing to find it in the text. More sophisticated NLP tools like NLTK can handle these complex scenarios (see [Chapter 16](#) on p. 298).

The counterpart of the `str.split()` function is `str.join(L)`. This function takes a list or a tuple of strings `L` and merges them into a single string, inserting the object string between each adjacent pair of strings from the list or tuple.

```
print(", ".join(students))  
#> Abby, Alice, Chuck  
print("| ".join(students))  
#> Abby| Alice| Chuck
```

10.1.2 Dictionaries

A dictionary represents another heterogeneous data container featuring enhanced indexing capabilities. The index of a dictionary item, also known as a key, is not restricted to sequential integer numbers. Instead, it can be a variety of data types, including strings, tuples, booleans, floating-point numbers, and even `None`, as long as each key in a dictionary is unique.

A dictionary is represented in Python by enclosing it within curly braces `{}`. Each dictionary item consists of a key and its corresponding value,

separated by a colon, with different items being divided by commas. Here is an example of a dictionary that holds student names and their ranks:

```
students_d = {"Alice" : "Freshman", "Bob" : "Sophomore",
              "Chuck" : "Senior"}
print(len(students_d))
#> 3
print(students_d["Alice"])
#> Freshman
```

Bear in mind that dictionary items do not have ordinal indexes. There is no “item number 0”!

You can access the lists of all keys, all values, and all items (as pairs of keys and values packaged in two-item tuples) by calling the namesake functions.

```
print(list(students_d.keys()))
#> ['Alice', 'Bob', 'Chuck']
print(list(students_d.values()))
#> ['Freshman', 'Sophomore', 'Senior']
print(list(students_d.items()))
#> [('Alice', 'Freshman'), ('Bob', 'Sophomore'),
#>  ('Chuck', 'Senior')]
```

Conversely, you can transform a list of two-item tuples into a dictionary using the function `dict()`—a dictionary constructor:

```
students_data = [('Alice', 'Freshman'), ('Bob',
                                         'Sophomore'), ('Chuck', 'Senior')]
students_d = dict(students_data)
print(students_d)
#> {'Alice': 'Freshman', 'Bob': 'Sophomore',
#>  'Chuck': 'Senior'}
```

Like lists, dictionaries are mutable. Items can be inserted, updated, or removed in a list-intuitive way, except the function `dict.append()` is called `dict.update()`.

```
students_d["Alice"] = "Sophomore" # Rising
students_d.update({"Alice": "Sophomore"}) # Same as above
students_d["Bob"] = "Junior" # Rising
students_d.update({"Bob": "Junior"}) # Same as above
del students_d["Chuck"] # Graduated
students_d["Abby"] = "Freshman" # New student
students_d.update({"Abby": "Freshman"}) # Same as above
```

In the field of NLP, Python dictionaries can be used, to a limited extent, as bilingual dictionaries, with words in one language being keys and the corresponding words in another language being values. Thus, you can split a text in one language into words, translate each word by looking it up in a dictionary,

TABLE 10.1
File input/output functions

<code>`TextIOWrapper.read()`</code>	Reads the whole file as one string
<code>`TextIOWrapper.read(n)`</code>	Reads up to <code>`n`</code> characters as a string
<code>`TextIOWrapper.readline()`</code>	Reads one line as a string
<code>`TextIOWrapper.readlines()`</code>	Reads all lines as a list of strings
<code>`TextIOWrapper.write(S)`</code>	Writes string <code>`S`</code>
<code>`TextIOWrapper.writelines(S)`</code>	Writes an array of strings <code>`S`</code>

and recombine the translations. However, you cannot expect a high-quality translation using this function.

10.2 Working with Files

Data for social science research predominantly comes from files. These files are typically organized according to pre-established data formats, such as CSV, JSON, or XML (see [Subsection 2.1.2](#) on p. 21). Specialized software is recommended to read these files to avoid programming errors and minimize potential complications. However, there may be instances where you come across a file in an unsupported format that you will need to parse manually.

10.2.1 Fundamentals of Text File Manipulation

Let’s assume that the file is a text file rather than a binary file¹. In that case, it is essentially non-volatile, long-term storage for a character string and needs two operations: writing a string to the file and reading a string from it.

For a file to be available for reading and writing, it has to be opened (for reading or writing, respectively) before the operation commences and closed after it completes. Luckily, Python’s context manager (employed through the ``with`` keyword) automatically manages the closing of open files. However, you still have to order them open. Once a file is opened, Python creates an opaque ``TextIOWrapper`` reference, which you store in a variable and later use for reading and writing. The reference becomes invalid when the file is closed. [Table 10.1](#) lists the basic file input/output functions.

If you suspect (or expect) that the file is large, you should avoid reading it in one piece, as it may not fit in your computer’s RAM. Reading files iteratively line by line is outside of this book’s scope.

¹To determine this, you can open the file in a basic text editor such as Notepad on Windows, TextEdit on MacOS, or Gedit on Linux. The file is likely a text file if its content appears legible and contains no “unusual” symbols.

Functions `TextIOWrapper.write(S)` and `TextIOWrapper.writelines(S)` do not add line breaks at the end of `S`. Breaking lines remains your responsibility.

Let's practice handling files with the example "data/CODE1:example.csv" (Listing 2.1 on p. 22). By judging its extension, you can tell it is a CSV file. You would commonly use Excel or Pandas to open such files, but let's explore the capabilities of raw Python.

First, call upon the context manager using the `with` keyword, instruct it to `open()` the file for reading (designated as `"r"`), and store the resulting file reference in the variable `infile` (or any other variable name of your choice). Then, read the file content into a separate variable, `data`. Be mindful of the indentation within the context manager's body; all indented code belongs to the context and can access the file.

The first `print()` line, not being indented, is outside of the context and can no longer access the file, but it can still utilize the data previously read from the file.

```
with open("data/example.csv", "r") as infile:
    # infile is a TextIOWrapper
    data = infile.read()
print(type(data))
#> <class 'str'> — a string!
print(len(data))
#> 95
lines = data.split("\n")
headers = lines[0] # The first line
print(headers)
#> Name, Gender, Rank, GPA
# The headers, split by commas
print(headers.split(","))
#> ['Name', 'Gender', 'Rank', 'GPA']
# The Chuck data, split by commas
lines[-2].split(",")
#> ['Chuck', 'Male', 'Senior', '3.00']
```

Naturally, you can handle an input file as plain text, disregarding any specific data format. For instance, you can break the text into individual words, count their occurrences, and report the most frequently appearing word by writing it into a new file. Listing 10.1 shows a complete program solving this problem.

Listing 10.1

most-common-word.py

```
1 # Find the most common word in one file
2 # and write it to another file
3 from collections import Counter
4
```



```

5 with open("data/example.csv", "r") as infile:
6     # infile is a TextIOWrapper
7     text = infile.read()
8
9 words = text.split()
10 counts = Counter(words)
11 result = counts.most_common(1)[0][0]
12
13 with open("data/example.dat", "w") as outfile:
14     # outfile is a TextIOWrapper
15     outfile.write(result + "\n")

```

The argument `"w"` stands for “writing.”

10.2.2 Pickling

Working directly with text and binary files is uncommon in social science research. You will learn about advanced file-handling techniques in [Section 10.3](#) on p. 197. As an introduction to higher-level data management, consider looking into pickling.

Pickling is a way to save and load your data. By using pickling, you can pause your work at any point, save it, and then resume later precisely where you left off, even if you have closed your Python environment. For instance, after preprocessing raw data or training a machine learning model, you could save (or “pickle”) them, then quickly load your saved data or model whenever you want to use them, conserving the time and resources it would take to preprocess or retrain them each time.

Pickling is supported by the module ``pickle`` that may need to be installed separately. Once imported, the module provides the functions ``pickle.dump(object, file)`` and ``pickle.load(file)``. Both require a reference to a file previously opened by a context manager in the binary mode (`"rb"` or `"wb"`). The ``object`` is any Python object, such as a number, a string, a list, a dictionary, a Pandas dataframe, or a machine learning model. Function ``pickle.dump(object, file)`` stores the object to a file, and function ``pickle.load(file)`` reads and returns the object.

```

import pickle
numbers = [1, 5, 32, 5, 8, 34, 67, 8, -7]
with open("temp.p", "wb") as outfile:
    pickle.dump(numbers, outfile)
# Two weeks later:
with open("temp.p", "rb") as infile:
    numbers1 = pickle.load(infile)
print(numbers == numbers1)
#> True

```

Unlike pickled fruit and vegetables, pickled data does not expire!

10.3 Pandas

Pandas is a specialized library designed for data manipulation and analysis in Python, specially tailored for labeled data. It introduces two primary data structures: the series and the dataframe. They resemble programmable spreadsheets with one (series) or multiple (dataframe) columns. Our primary focus will be on the multi-column dataframes, with attention to series only as needed.

Pandas facilitates the import, cleaning, transformation, and analysis of data. It allows you to filter rows, select specific columns, merge datasets, group data by selected variables, and compute descriptive statistics. Moreover, it integrates flawlessly with other Python libraries like NumPy for mathematical computations, Matplotlib for data visualization, sklearn for machine learning, and SciPy for statistical analysis.

Pandas is a third-party module and may need to be installed separately, often alongside NumPy and Matplotlib (see [Subsection 9.3.1](#) on p. 186 to refresh your module installation skills).

Pandas is usually imported as ``import pandas`` or ``import pandas as pd``. The latter form is considered more idiomatic.

You can initialize a dataframe/series in three ways:

- Manually create an empty dataframe/series and populate it with data. This technique is the least common one.
- Convert an existing list of lists, a list of dictionaries, or a dictionary of lists into a dataframe; convert an existing list or dictionary into a series. This function offers an interface between traditional data types and Pandas.
- Import data directly from databases (both SQL and NoSQL), files (in formats like CSV, XML, JSON, Excel), or Web APIs. This efficient and time-saving option is the focus of our subsequent discussion.

10.3.1 Importing Data

Importing data into Pandas from a well-structured file in a widely supported format is straightforward. All you need to do is invoke the function that matches the data format, such as ``pandas.read_csv()`` for CSV, ``pandas.read_json()`` for JSON, ``pandas.read_xml()`` for XML (available in Pandas 1.30 and newer), and ``pd.read_excel()`` ``pandas!read_excel()`` for an Excel or OfficeLibre file. The last function, by default, reads only the first sheet but recognizes the files with the extensions “xls,” “xlsx,” “xslm,” “xlsb,” “odf,” “ods,” and “odt.” Besides, Pandas also supports numerous other data formats, including SAS, SQL, SPSS, and HTML ([Section 16.2](#) on p. 301 explains the latter in detail). The contents of the files are shown on p. 22.

```
import pandas as pd
df_csv = pd.read_csv("data/example.csv")
df_jsn = pd.read_json("data/example.json")
df_xml = pd.read_xml("data/example.xml")
```

Notably, each function requires only a single argument: the file name. You are not required to set up an I/O context, open or close the file, or even read it as text line by line or as a whole. The functions handle the heavy lifting, including file management, data conversion, error checking, date and time recognition, and column and row label extraction, leaving the fun part, the data analysis, to you.

You do you confirm if the data was correctly imported? Firstly, the absence of error messages suggests that the process was successful.

Next, the `len()` function, the “universal yardstick,” reveals that each dataframe contains three rows. Additionally, each dataframe exhibits a shape of `(3,4)` (a tuple denoting three rows and four columns) and contains $3 \times 4 = 12$ items. `DataFrame.size` and `DataFrame.shape` are dataframe attributes. Attributes are variables that cannot be invoked.

```
print(len(df_csv), len(df_jsn), len(df_xml))
#> 3 3 3
print(df_csv.shape, df_jsn.shape, df_xml.shape)
#> (3, 4) (3, 4) (3, 4)
print(df_csv.size, df_jsn.size, df_xml.size)
#> 12 12 12
```

The three dataframes are not only the same size but also have the same values². Therefore, you can select any dataframe for further examination and label it `df`, an idiomatic name when dealing with a single dataframe in your program.

Among all the functions like `pandas.read...`, `pandas.read_csv()` has a distinct ability to recognize missing values, which may be represented in forms such as an empty string, N/A, NaN, nan, <NA>, NULL, null, or n/a, among others. The function also offers an optional argument, `na_values`, which lets you specify additional symbols or values that should be recognized as missing data. This argument can accept a constant, a list, or a dictionary. Using a dictionary lets you specify different missing value designators for different columns. Early detection of missing values is crucial in preventing inaccuracies in subsequent computations.

²Not exactly. Due to the inaccuracy of floating-point numbers (p. 185), Alice’s GPA is 3.82 in the CSV dataframe, but 3.8200000000000003 in the JSON dataframe.

10.3.2 Accessing Rows and Columns

A newly created dataframe in Pandas resembles a spreadsheet consisting of four rows (including a row for column labels) and five columns (including a column for row labels).

```
print(df)
#>      Name  Gender      Rank  GPA
#> 0  Alice  Female  Freshman  3.50
#> 1   Bob   Male   Sophomore  3.82
#> 2  Chuck   Male    Senior   3.00
```

The row labels (ranging from zero to two) form the row index, while the column labels make up the column index. These indexes allow you to select individual rows, columns, slices, or items from a dataframe, which you will explore later.

By default, Pandas assumes that the first line of a CSV file represents the column index (though the scenario is more complex with JSON and XML files). This assumption can be overwritten by passing the option ``header=None``.

```
df_csv = pd.read_csv("data/example.csv", header=None)
print(df_csv)
#>      0      1      2      3
#> 0  Name  Gender      Rank  GPA
#> 1  Alice  Female  Freshman  3.50
#> 2   Bob   Male   Sophomore  3.82
#> 3  Chuck   Male    Senior   3.00
```

Consequently, the dataframe will have a positional column index, with the first row as a valid data sample. The row index is positional by default, but you can designate any column to serve as the index—for instance, the “Name” column.

```
df_csv = pd.read_csv("data/example.csv", index_col="Name")
print(df_csv)
#>      Gender      Rank  GPA
#> Name
#> Alice  Female  Freshman  3.50
#> Bob    Male   Sophomore  3.82
#> Chuck  Male    Senior   3.00
```

Once designated, the column is moved from the dataframe body to the far-left position, and its name is placed on the second line to highlight its special role. It is important to note that despite these modifications, the dataframe still contains only three data rows!

Numerous other optional arguments allow you to fine-tune the importation process.

You can retrieve the row index with `DataFrame.index` and the column index with `DataFrame.columns`. Both attributes are of the `Index` data type, and while they can be manually modified, it is generally not advisable.

```
print(df.columns)
#> Index(['Gender', 'Rank', 'GPA', ('GPA', 'Alice')],
#>      dtype='object')
print(df.index)
#> Index(['Alice', 'Bob', 'Chuck'], dtype='object', name='Name')
print(df.index.name)
#> Name
```

The row index of a dataframe can be reconfigured at any point using the functions `DataFrame.reset_index()` and `DataFrame.set_index(name)`. The `DataFrame.reset_index()` function moves the existing index into the body of the dataframe and re-establishes the default positional index. On the other hand, the `DataFrame.set_index(name)` function substitutes the current index with the specified column.

```
df = df_csv.reset_index().set_index("Rank")
print(df)
```

	Name	Gender	GPA
#> Rank			
#> Freshman	Alice	Female	3.50
#> Sophomore	Bob	Male	3.82
#> Senior	Chuck	Male	3.00

Data analysis often involves accessing the entire dataframe and its parts: rows, columns, and items. Rows and columns might need to be selected individually, as slices, or as non-contiguous sets. Like NumPy, Pandas supports four indexing modes: direct, slices, boolean indexing, and smart indexing. Each mode applies to both positional and label-based indexes.

To access a single column by its name, treat the dataframe like a dictionary ([Subsection 10.1.2](#) on p. 192) and use the selection operator `[]`. The result is a series—fitting, as a dataframe can be considered a dictionary of series!

```
print(df["GPA"])
#> Name
#> Alice      3.50
#> Bob        3.82
#> Chuck      3.00
#> Name: GPA, dtype: float64
```

The name of the series is `'GPA'`, identical to the column's name, and its data type is `'float64'`, a floating-point number. The series retains the same row index as the original dataframe. This index can select an item from the series, treating the series as a dictionary—since a series often mimics dictionary behavior!

TABLE 10.2
Dataframe accessors

Accessor	Row(s)	Column(s)
Direct accessors		
<code>`df.loc["Alice"]`</code>	"Alice"	...
<code>`df.loc[:, "GPA"]`</code>	...	"GPA"
<code>`df.loc["Bob":, "GPA"]`</code>	"Bob"	"GPA"
<code>`df.iloc[0]`</code>	...	#0
<code>`df.iloc[:, 1]`</code>	...	#1
<code>`df.iloc[0, 1]`</code>	#0	#1
Slices		
<code>`df.loc["Bob":]`</code>	"Bob", ...	...
<code>`df.loc[:, "Rank":]`</code>	...	"Rank", ...
<code>`df.loc[:"Bob", "Rank"]`</code>	..., "Bob"	"Rank"
<code>`df.loc["Alice", : "Rank"]`</code>	"Alice"	..., "Rank"
<code>`df.loc[:"Bob", "Rank":]`</code>	..., "Bob"	"Rank", ...
<code>`df.iloc[1:-2]`</code>	1, ..., -2	...
<code>`df.iloc[:, :2]`</code>	...	0, ..., 2
<code>`df.iloc[1:, :2]`</code>	1, ...	0, ..., 2

```
print((df["GPA"])[ "Alice" ])
#> 3.5
```

The inner parentheses are unnecessary; they underscore the selection from a series.

You can even change Alice’s grade if you believe she deserves more (or less):

```
df["GPA"][ "Alice" ] = 3.7
print(df["GPA"][ "Alice" ])
#> 3.7
```

Accessors

For other indexing operations, you should utilize accessors ``DataFrame.loc[]`` (locations by labels) and ``DataFrame.iloc[]`` (locations by positions). These accessors are dataframe attributes that operate on column and row indexes. For example, ``df.loc["Alice"]`` means “access the row locations of Alice for all columns.” Table 10.2 shows a plethora of examples of direct accessors and slices. The comma within the square brackets differentiates between row and column access. Beware that you cannot mix access by labels and positions.

You are not restricted to selecting only one column/row or a contiguous slice of them; you can also select a set of columns/rows.

Pandas allows you to pass a list of labels to the `DataFrame.loc[]` accessor or a list of positions to the `DataFrame.iloc[]` accessor. An intriguing aspect of this “smart indexing” is that labels or positions can be included on the list in any order and possibly repeated. Let’s consider an unconventional example to illustrate this.

```
>>> df.loc[["Alice", "Chuck"], ["GPA", "GPA", "Rank"]]
#>      GPA  GPA      Rank
#> Name
#> Alice  3.7  3.7  Freshman
#> Chuck  3.0  3.0   Senior
```

The final indexing mode is called boolean indexing. Instead of specifying labels or positions, you create a list or a series whose boolean values are `True` for the columns/rows to be selected and `False` for those to be excluded. Its length must match the number of columns/rows. For example, a list could reflect a certain characteristic, such as whether a student is a freshman or if their GPA is below 3.0. You will learn how to construct such a series in 10.3.5 on p. 207. Let’s assume we want to compile a dataframe of male students and that the boolean list has been created—perhaps manually.

```
is_male = [False, True, True]
selection = df.loc[is_male]
print(selection)
#>      Gender      Rank  GPA
#> Name
#> Bob      Male  Sophomore  3.82
#> Chuck    Male   Senior   3.00
```

Boolean indexing works with and without accessors, but the list or the series must be entirely boolean. If it contains a single non-boolean value, it will be interpreted as a container of integers and select different results, as illustrated in the following example:

```
print(df.iloc[[0, 1, 1]])
#>      Gender      Rank  GPA
#> Name
#> Alice  Female  Freshman  3.70
#> Bob      Male  Sophomore  3.82
#> Bob      Male  Sophomore  3.82

print(df.iloc[[False, True, True]])
#>      Gender      Rank  GPA
#> Name
#> Bob      Male  Sophomore  3.82
#> Chuck    Male   Senior   3.00
```

The first expression selects rows #0 (once) and #1 (twice). The second expression selects rows #1 and #2 (because the corresponding list items are `True`).

10.3.3 Data Types and Levels

Given the previous example, you might be curious about how to convert a list intended for boolean indexing to the appropriate data type. More broadly, how can one convert a container from one data type to another?

This operation, known as type conversion, is not always feasible. First, converting the data type of lists and dictionary values requires understanding loops, which you will explore only in [Section 12.1](#) on p. 234. Fortunately, lists are rarely used for boolean indexing, and series do not require loops.

The second problem is more fundamental. Take, for example, the integer number 314. You can convert it to a string by calling ``str(314)``. This conversion applies to any integer. The resultant string "314" can be converted back to the same number by calling ``int("314")``. This conversion has a limited domain: many strings (indeed, most) do not resemble numbers and thus cannot be converted. To summarize, a series of integers can be transformed into a series of strings, but a series of strings generally cannot be turned into a series of integers.

To convert a series or a dataframe column (also a series) to a different data type, use the function ``Series.astype(new_type)``.

```
print(df["GPA"].astype(int))
#> Name
#> Alice      3
#> Bob        3
#> Chuck      3
#> Name: GPA, dtype: int64
print(df["GPA"].astype(bool))
#> Name
#> Alice      True
#> Bob        True
#> Chuck      True
#> Name: GPA, dtype: bool
print(df["Gender"].astype(int))
#> Traceback (most recent call last):
#> ...
#> ValueError: invalid literal for int() with base 10: 'Female'
```

The function creates and returns a new series if the conversion is viable. Otherwise, it signals an error. You can store the new series in a variable or reinsert it into the original dataframe, irreversibly overwriting the old values of the converted column.

```
df["GPA"] = df["GPA"].astype(int)
```

A common misstep in data analysis is presuming that values appearing numerical are indeed numbers. They might be strings and, therefore, unfit for arithmetic operations. To verify the data type of a column, refer to the

code listing on p. 200. If it does not align with your expectations, invoke the conversion function.

Another type of conversion commonly encountered in social science research is categorization: transforming a continuous variable into discrete levels of measurement, also known as categories or bins ([Subsection 2.3.1](#) on p. 25).

For the remainder of this Chapter, you will engage with the dataset from the World Happiness Report [[HLS+22](#)], particularly exploring the variable “Generosity.” Also, unless otherwise stated, the data will be from 2021.

```
happy = pd.read_csv("data/happiness.csv", index_col=0)
happy21 = happy[happy['year'] == 2021]
generosity = happy21["Generosity"]
```

Two powerful Pandas functions, ``pandas.cut()`` and ``pandas.qcut()``, offer a robust and helpful approach to categorizing data. These functions partition the range of the variable into intervals (or bins) of varying or uniform width and assign categorical labels (levels of measurement) to the values in each interval. You can let these labels be generated automatically or provide them for improved readability and interpretation. The bins can also be defined automatically based on the desired number N , or you can define them yourself.

The function ``pandas.cut()`` operates assuming the bins have the same width $w_x = (\max x - \min x)/N$. This function essentially mirrors the behavior of ``Series.hist()``. In the following example, we define a tuple of three labels: “Selfish,” “Neutral,” and “Generous.” Each value, except the missing values (p. 198), is then assigned to one of these three bins, equivalent to assigning a label to each value.

```
labels = ("Selfish", "Neutral", "Generous")
c_bins = pd.cut(generosity, len(labels), labels=labels)
print(c_bins)
#> 12      NaN
#> 26      Neutral
#> 36      Selfish
#> 56      Selfish
#> 71      Selfish
#> ...
#> 2013     Neutral
#> 2029      NaN
#> 2045     Neutral
#> 2072     Neutral
#> 2088     Selfish
#> Name: Generosity, Length: 119, dtype: category
#> Categories (3, object): ['Selfish' < 'Neutral' < 'Generous']
```

This series can be incorporated into the original dataframe as a column ([Subsection 10.3.4](#) on p. 205) or used as a standalone variable. If you choose the latter, you may want to perform a count of how many values belong to each

level. The function ``Series.value_counts()`` returns a new series of counts, sorted in descending order. Combined with the functions ``Series.idxmax()`` and ``Series.max()``, it allows you to determine the most frequently occurring level and its frequency.

```
counts = c_bins.value_counts()
print(counts)
#> Neutral      58
#> Selfish      47
#> Generous       6
#> Name: Generosity, dtype: int64
print(counts.max()) # The largest value
#> 58
print(counts.idxmax()) # The label of the largest value
#> Neutral
```

The function ``pandas.qcut()`` acts similarly to ``pandas.cut()``, but it equalizes the population within the bins rather than their widths. Consequently, you end up with the same three levels, but each level has an equal or approximately equal number of values. This function proves particularly useful when dealing with skewed data.

```
q_bins = pd.qcut(generosity, len(labels), labels=labels)
print(q_bins.value_counts())
#> Generous      37
#> Selfish       37
#> Neutral       37
#> Name: Generosity, dtype: int64
```

10.3.4 Row and Column Manipulation

A dataframe, as a data container, is designed to be dynamic, offering mechanisms for adding, reordering, and removing columns and rows.

Inserting rows is rare. If you have two dataframes that need to be combined, you should typically use functions like ``pandas.merge()`` or ``pandas.concat()`` (10.4.1 on p. 211). However, if row insertion is necessary, the ``DataFrame.append(X)`` function can add a dataframe, series, or dictionary ``X`` to the end of the object dataframe, updating the index. To illustrate, let's create a new series from a dictionary, assign it a name that will act as the new index label, and form a new dataframe with the new record.

```
zack_data = pd.Series({"Gender": "Male"}).rename("Zack")
df_with_zack = df.append(zack_data)
print(df_with_zack)
#>      Gender      Rank  GPA
#> Name
#> Alice  Female  Freshman  3.70
```

```
#> Bob      Male  Sophomore  3.82
#> Chuck    Male    Senior    3.00
#> Zack     Male      NaN     NaN
```

The appended series is intentionally incomplete, which results in the dataframe having some missing values denoted as ``NaN`` (“not a number”). While ``NaN`` technically behaves as a floating-point number, most arithmetic operations involving it will also result in ``NaN``. Depending on which modules you have imported, there are a few ways to generate ``NaN``.

```
float("NaN") # The Python way
#> nan
import math # The math way
math.nan
#> nan
import numpy as np # The NumPy way
np.nan
#> nan
```

[Subsection 10.3.6](#) on p. 208 explains how to handle unwanted ``NaN``s.

On the contrary, adding a column is a frequent and helpful operation. A column in a dataframe typically represents an attribute of the dataset. New attributes can be sourced from other dataframes and series or calculated as secondary data—e.g., by categorizing a continuous variable or merging date and time columns into a `DateTime` column.

Adding a new column into an existing dataframe is straightforward through assignment, directly or via an accessor.

```
happy21["Generosity level"] = c_bins # Way 1
happy21.loc[:, "Generosity level"] = c_bins # Way 2
```

Before the column’s insertion, the assignment operator aligns the dataframe’s index with that of the column. This process, known as data alignment, positions a new value with the label `X` in an existing row with the corresponding label, disregarding row numbers. If the indexes do not share the same labels, non-matching rows are populated with missing values.

To drop a column or a row, you would use the ``DataFrame.drop()`` function along with the label or list of labels you wish to drop. The keywords ``columns`` and ``index`` indicate which index is affected.

```
happy21.drop(columns=["Generosity level"], inplace=True)
df.drop(index=["Chuck"], inplace=True)
```

Unlike all other functions you have encountered, ``DataFrame.drop()`` allows for modifying the dataframe in place (``inplace=True``) or creating a modified copy of the original dataframe. While the in-place modification is memory-efficient, particularly for large datasets, it is also irreversible. On the

other hand, returning a modified copy can be memory-intensive. You should choose the best method for your research objectives and available resources.

10.3.5 Filtering

Dropping a row serves as a data filtering mechanism, eliminating unwanted rows—such as those containing missing values or failing to meet specific criteria. Conversely, another form of data filtering involves retrieving rows that do meet particular conditions and placing them into another dataframe for subsequent analysis. The choice between dropping and retrieving ultimately depends on the quantity of the affected rows, but regardless of the choice, you need a series for boolean indexing.

Understanding Pandas’ structure and hierarchy can assist in preparing the series. A Pandas series is essentially a NumPy array: a homogeneous list wherein all items share the same data type. These arrays can exist in one or more dimensions. A one-dimensional, linear array, often termed a vector, is sufficient for our purposes. To summarize:

1. A dataframe is comprised of columns.
2. Each column is a series.
3. Each series is a NumPy vector with a custom index.
4. An array presents a list-like interface.

Both arrays and series support vectorized logical, relational, and arithmetic operators ([Subsection 9.2.4](#) on p. 181). These operators function simultaneously across all vector items. For instance, the expression `-score` negates all items within the `score` series, while `date+period` calculates the element-wise sums of `date` and `period`. Similarly, the expression `age>limit` generates a boolean value for each pair of aligned ages and limits. Furthermore, NumPy offers a library of vectorized functions, such as `numpy.sin(x)` and `numpy.log(x)`, which operate on all items of `x`. For instance, the “log plus 1” transformation of `x` ([Subsection 2.3.5](#) on p. 30) can be expressed in Pandas as `np.log(x+1)`.

One notable exception is the logical operators `and`, `or`, and `not`. The vectorized versions of these operators are represented as `&`, `|`, and `~`, respectively, and should be parenthesized to ensure proper precedence (p. 9.2.4).

Identifying specific conditions within your data—particularly tailored to your research—requires translation into a Pandas expression: utilizing vectorized operators and functions and treating the columns as variables. As a result, the expression should yield a boolean series that can then be used for indexing and row dropping or retrieval. Let’s consider a few examples to clarify this process:

- Select all rows that belong to the “Selfish” generosity level group:

```
condition = happy21["Generosity level"] == "Selfish"
result = happy21[condition]
print(result.shape[0])
#> 47
```

- Select all rows for which the positive affect is three or more times higher than the negative affect:

```
condition = happy21["Positive affect"] >= \
            3 * happy21["Negative affect"]
result = happy21[condition]
print(result.shape[0])
#> 30
```

- Select all rows that do not belong to the “Selfish” group and relate to 2021:

```
condition = (happy21["Generosity level"] != "Selfish") & \
            (happy21["year"] == 2021)
result = happy21[condition]
print(result.shape[0])
#> 64
```

- Select the opposite of the previous condition:

```
result = happy21[~condition]
print(result.shape[0])
#> 55
```

The `result` is a subset of the original dataframe. You can obtain its index and use it to drop the matching rows:

```
remaining = happy21.drop(index = result.index)
print(remaining.shape[0])
#> 64
```

A filtered dataframe is still a dataframe. You can use it for further operations.

10.3.6 Handling Missing Values

Missing data presents significant challenges in data analysis ([Subsection 2.3.4](#) on p. 29). These missing values can arise due to incomplete data collection, data aggregation, and removal of sensitive data, among other reasons. Recognizing and handling these missing values is a critical aspect of data analysis.

As outlined on p. 198, missing values are ideally detected when importing data into a dataframe. Regardless of the initial representation, Pandas encodes them as `NaN` and expects you to follow this convention.

Several strategies can be adopted to deal with missing values:

1. Ignore them for the time being and handle them later—an *ad hoc* strategy.
2. Discard them. The `DataFrame.dropna()` function allows you to drop a row (`axis=0`) or a column (`axis=1`) containing at least one (`how="any"`) or all (`how="all"`) missing values. Additionally, the function supports the `inplace` argument, which modifies the dataframe in place. Thus, `happy21.dropna(axis=0, how="any", inplace=True)` would eliminate all rows with at least one missing value in place, directly altering the dataframe. However, this method might be too aggressive if your dataframe contains numerous scattered missing values.
3. Substitute the missing values with other values, a process known as imputation, which is further explained below.

Data Imputation

Functions `DataFrame.isna()` and `DataFrame.isnull()` can be used to generate a boolean dataframe or series (depending on what is `D`) that indicate the location of missing values in the original container. Conversely, `DataFrame.notna()` and `DataFrame.notnull()` perform the opposite operation, indicating where values are present³.

Some common data imputation methods include substitution with a constant (default value, mean, median, or mode) and forward/backward fill.

The boolean dataframe or series of missing data locations mentioned above can be used as a boolean index. Consider the dataframe with partially valid data on p. 205. It has a missing GPA (a number) and a missing rank (a level of measurement). You can replace the missing GPAs with some constant default value:

```
missing_gpa = df_with_zack["GPA"].isnull()
DEFAULT_GPA = 3.0 # Use all upper-case letters for constants!
df_with_zack.loc[missing_gpa, "GPA"] = DEFAULT_GPA
```

After imputation, the dataframe may seem complete, but remember that some data points are synthetic. It is crucial to be aware of this, as it can influence your subsequent analysis and interpretations.

You will learn about a more robust constant imputation technique in [Subsection 11.4.1](#) on p. 227.

³The four functions are identical pairwise. Pandas inherited them from the R language dataframes that distinguish between a `NULL` object and a missing value indicator `NA`.

Another imputation method is often applicable to time series data, such as stock quotes or population growth, because one can reasonably assume that such data will exhibit smooth, incremental changes over time rather than abrupt jumps.

For these types of data, forward filling (via the `DataFrame.ffill()` function) and backward filling (via `DataFrame.bfill()`) can effectively impute missing values. Forward filling involves filling a missing value with the most recent known value before the missing point. Conversely, backward filling uses the first known value following the missing point. By design, `DataFrame.ffill()` does not fill the first row, and `DataFrame.bfill()` does not fill the last row. To address this and potentially create more accurate imputations, you could use both methods and then average the results. This approach provides a more balanced estimate by considering information from both directions in the time series.

```
peru_ladders = happy.loc[happy["Country name"] == "Peru",
                        ["year", "Life Ladder"]]\
                        .set_index("year")
corrupted = peru_ladders.copy()
corrupted.loc[[2012, 2013, 2015, 2018, 2019]] = np.nan
imputed = (corrupted.ffill() + corrupted.bfill()) / 2
```

Finally, you can interpolate over missing time series data using `Series.interpolate()`. The function provides many configuration options and uses SciPy, a statistical analysis module, meaning SciPy must be installed.

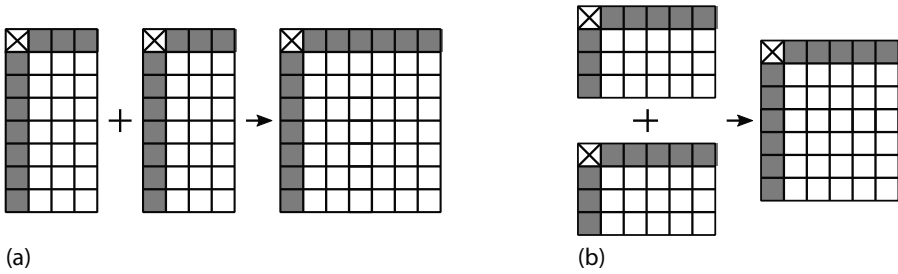
```
interpolated = corrupted.interpolate()
```

10.4 Combining Dataframes

Social science datasets in real-world scenarios rarely exist as a single, complete set. Often, various aspects of data—such as demographic, economic, political, and cultural—need to be collected from disparate sources and consolidated for further research. For instance, consider a dataframe on happiness that does not include details about income groups, country codes, and regions. Country codes serve as standardized identifiers for countries and territories, while regions are typically used to investigate mesoscopic phenomena.

However, the file “data/cc-region.csv” contains a dataset that includes country codes, regions, and country names (listed as “TableName”) alongside several unrelated columns.

```
cc = pd.read_csv("data/cc-region.csv", index_col="Country Code")
```

**FIGURE 10.1**

Horizontal (a) and vertical (b) concatenation.

```
print(cc.columns)
#> Index(['Region', 'IncomeGroup', 'SpecialNotes', 'TableName'],
#>        dtype='object')
```

10.4.1 Merging and Joining

Pandas offers several functions to help consolidate related dataframes. The function `pandas.concat()` concatenates several dataframes vertically (by stacking) or horizontally (by placing them side by side). Stacking combines dataframes with the same columns (attributes) but different indexes (observations), while adjacent placement combines dataframes with the same index but different attributes (Figure 10.1).

The functions `DataFrame.merge()` and `DataFrame.join()` combine two dataframes side by side. Due to its flexibility, `DataFrame.merge()` can execute nearly all operations performed by `DataFrame.join()`, so the focus will primarily be on `DataFrame.merge()`.

The `DataFrame.merge()` function consolidates two dataframes based on columns with identical names, such as “Country Name.” Both dataframes’ rows are aligned in relation to this column or set of columns, as detailed on page 206. Subsequently, rows with matching values are concatenated horizontally. The columns from the first and second dataframes are copied into the new dataframe. Unless they possess different names, the duplicate columns used for merging are deduplicated.

If the names of the columns intended for merging differ, they must be specified using the keyword arguments `left_on` and `right_on` for the first and second dataframes, respectively. If the attributes intended for merging are the indices, they should be specified using the boolean keyword arguments `left_index` and `right_index`, setting them to `True` to use the index instead of a column. The process of merging on indices is actually referred to as joining.

Let’s merge the pieces and eliminate the redundant “TableName” column:

```
happy_full = happy.merge(cc, left_on="Country name",
```



```

right_on="TableName")\
.drop(columns="TableName")

print(happy_full.columns)
#> Index(['Country name', 'year', 'Life Ladder',
#> 'Log GDP per capita', 'Social support',
#> 'Healthy life expectancy at birth',
#> 'Freedom to make life choices', 'Generosity',
#> 'Perceptions of corruption', 'Positive affect',
#> 'Negative affect', 'Confidence in national government',
#> 'Region', 'IncomeGroup', 'SpecialNotes'], dtype='object')
happy_full21 = happy_full[happy_full["year"] == 2021]

```

Then, you can save and use the new dataframe instead of the original two dataframes.

```
happy_full.to_csv("data/happiness-full.csv")
```

10.4.2 Merging Partially Matching Dataframes

The process of merging dataframes proceeds seamlessly when the column used for merging shares identical items across both dataframes. However, complications can arise when it is not the case.

Take, for instance, a situation involving two dataframes: one, `df_csv`, comprising student demographics (Listing 10.3.2 on p. 199) and the other consisting of their school ID numbers. The first dataframe includes rows for Alice, Bob, and Chuck, while the second mentions Dinah but omits Alice.

```

df_csv = pd.read_csv("data/example.csv")
df_sid = pd.read_csv("data/example-id.csv", dtype={"SID": str})
joined = df_csv.merge(df_sid)
print(joined)
#>   Name Gender      Rank  GPA    SID
#> 0   Bob   Male  Sophomore  3.82 1234331
#> 1  Chuck   Male    Senior  3.00 5567615

```

Although the resulting operation is accurate, it is incomplete. Alice is absent due to a missing school ID, and Dinah is missing because her demographics are not included. This process of consolidating two dataframes while eliminating non-matching rows is known as an “inner join.” An inner join ensures that no new missing values are introduced.

However, if you aim to preserve as much data as possible, addressing missing values later, an “outer join” is more appropriate. This method inserts missing values as required, potentially combining two complete datasets into one that is incomplete.

```

joined_outer = df_csv.merge(df_sid, how="outer")
print(joined_outer)

```

```
#>      Name Gender      Rank GPA      SID
#> 0  Alice  Female  Freshman 3.50      NaN
#> 1   Bob   Male   Sophomore 3.82 1234331
#> 2  Chuck   Male     Senior 3.00 5567615
#> 3  Dinah   NaN      NaN    NaN 0019328
```

Some integer and boolean columns might be transformed into floating-point columns as `NaN` is a floating-point value, and all values in a single column must share the same data type.

It is also possible to retain non-matching rows exclusively from the left dataframe (`how="left"`) or from the right dataframe (`how="right"`), though such scenarios are rare.

10.4.3 Pivoting and Stacking

[Section 2.2](#) on p. 23 introduced the concept of wide and long data formats. In data analysis, you can manipulate the number of rows and columns to make your dataframe taller (“longer”) or wider. Pandas supports such conversion between these data arrangements.

One of the tools provided is the function `DataFrame.pivot()`. The function “pivots” the dataframe by converting the value of one of the selected columns into new column headers.

Consider the dataframe of interstate migrations in the USA in 2021 from the Internal Revenue Service (IRS) website, available in “data/migrations2021.csv”. The dataframe contains 2601 rows (51 × 51 records for each state and the District of Columbia, DC) and five columns: the two-letter destination and origin states codes, number of migrated tax returns (essentially, households), number of migrated individuals, and the migrated adjusted gross income (AGI) in thousands of dollars.

```
migration = pd.read_csv("data/migrations2021.csv", index_col=0)
print(migration.shape)
#> (2601, 5)
print(migration.columns)
#> Index(['Destination', 'Origin', 'N_Returns', 'N_Individuals',
#>        'AGI'], dtype='object')
```

The database is in a long format, with one row for each pair of states. Here is how you look up migrations between Massachusetts and DC (both ways):

```
states = ("MA", "DC")
print(migration[(migration["Destination"].isin(states)) &
                 (migration["Origin"].isin(states))])
#>      Destination Origin  N_Returns  N_Individuals      AGI
#> 451           DC     DC    258325.0      420907.0  31591993.0
#> 460           DC     MA      805.0        1007.0    99838.0
#> 1179          MA     MA   2660273.0    5066625.0  314535057.0
```

#> 1202	MA	DC	985.0	1143.0	186795.0

We use the `Series.isin(c)` function that checks if an item of the series is in the container `c`. The solution is correct but not quite humane. Wouldn't it be nice to have destination states as the row index and origin states as the column names—in other words, make the dataframe “square” by converting it into a wide format? That is what `DataFrame.pivot()` does!

- Choose “Destination” as the new row index (the old default index will be discarded, but you do not need it);
- Choose “Origin” as the new column index;
- Choose one of the remaining columns, e.g., “N_Individuals,” as the table values, and call the function.

```
migration_mtx = migration.pivot("Destination", "Origin",
                                "N_Individuals")
```

The new wide dataframe `migration_mtx` may have missing values even if the long one did not due to missing lines in the latter. These missing values represent the absence of migration flows and should be set to 0. As a result, all dataframe values are now integers, and you can use `.loc` to access them.

```
migration_mtx = migration_mtx.fillna(0).astype(int)
print(migration_mtx.shape)
#> (51, 51)
>print(migration_mtx.loc[states, states])
#> Origin      MA      DC
#> Destination
#> MA          5066625    1143
#> DC          1007    420907
```

The inverse operation, transforming a wide dataframe into a long one, does not exist in Pandas in one piece but is still available as a combination of `DataFrame.stack()` and `DataFrame.reset_index()`.

Function `DataFrame.stack()`, as its name suggests, stacks all columns on top of each other, with the first column at the top and the last column at the bottom⁴. The function also builds a second-level index that converts column names into unique row labels for each stacked column. For example, the column name “MA” becomes the second-level row label for each column item in addition to the original index names. The item `migration_mtx.loc["MA", "DC"]` becomes known as `migration_mtx.stack().loc[("MA", "DC")]` in the stacked long dataframe, where the tuple `("MA", "DC")` is a two-level index⁵ of the value.

⁴This description applies only to dataframes with flat indexes. All dataframes considered in the book so far have flat indexes.

⁵An index with more than one level is called a MultiIndex.

```
print(migration_mtx.stack().loc[("MA", "DC")])
#> 1143
print(migration_mtx.stack().loc[("DC", "MA")])
#> 1007
```

An index can be re-incorporated into the dataframe with a call to `DataFrame.reset_index()`, as explained in [Subsection 10.3.2](#) on p. 199. As a result, you have the original dataframe (save the last column name):

```
print(migration_mtx.stack().reset_index())
#>      Destination Origin      0
#> 0              AK     AK 527600
#> 1              AK     AL   550
#> 2              AK     AR   277
#> 3              AK     AZ  1704
#> 4              AK     CA  1996
#> ...           ...     ...     ...
#> 2596           WY     VT    29
#> 2597           WY     WA   594
#> 2598           WY     WI   300
#> 2599           WY     WV    68
#> 2600           WY     WY 439711
```

10.5 What's Next

You have started with one or more separately obtained data files, cleaned them, took care of the missing values, and combined the tables into one table. The next Chapter introduces tools for descriptive and exploratory analysis of the table and simple data visualization: histograms and bar charts.

10.6 Programming Exercises

Exercise 5. Using functions `str.split()`, `str.join()`, and `sorted()`, write a code fragment that prints any text (e.g., the `text` variable) as a sorted list of words one word per line. Assume that a word is a substring without spaces.

Exercise 6. Write a program that asks the user to enter a small text file name. The program then reads the text from the file, converts it to the upper

case, and writes to a new file. The new file name is the same as the source file but has the prefix “upper-” appended before it.

Exercise 7. Using the function `Series.value_counts()`, find out what is the most frequently mentioned region in the `happy_full121` dataframe.

Exercise 8. Using the function `pandas.qcut()`, calculate the quartiles for the “Generosity” column of the `happy21` dataframe.

Exercise 9. What happens if the data type (`dtype`) of the column “SID” in Listing 10.4.2 on p. 212 is not specified?

Condition Checking and Descriptive and Exploratory Analysis

Data exploration is an inherently iterative process fraught with potential pitfalls. As a social science researcher, you may make mistakes like formulating unsound hypotheses, gathering inaccurate data, choosing unrepresentative samples, or constructing faulty data models. Moreover, as a Python programmer, you might craft code that fails to do what you intend, incorrectly assumes something about the data, falls short of accuracy expectations, or suffers from syntactic errors. It is crucial to anticipate these errors to avoid them or recover effectively when they inevitably occur.

As your programs become increasingly sophisticated, you need new tools to manage their complexity—in the first place, checking conditions and handling errors.

11.1 Condition Checking and Decision-Making

In [Subsection 9.2.4](#) on p. 181, you learned how to compare numbers and strings for equality and inequality using relational operators, such as `>=` (greater than or equal to) and `==` (equal). These operations yield boolean values, `True` and `False`, which can then be manipulated with logical operators: `not`, `and`, and `or`.

Consider the variables `pop` (population) and `gvt_type` (government type); you might want to determine if a country's population is not less than 22 million and its government type is a monarchy:

```
condition = (pop >= 22_000_000) and (gvt_type == "monarchy")
```

The boolean value of `condition` will depend on the variables unknown when writing the program. Subsequent steps in your program could then be determined based on the outcome of this condition. For instance, you might only want to analyze a dataset if the condition is met.

Python provides a conditional statement that allows for this kind of decision-making. This statement selects one of two possible actions based on the condition's boolean value. In its simplest form, it executes a block of code

(the “body” of the statement) if the condition is true; if not, it bypasses the body and continues with the next statement. Each line of the body is indented uniformly, and a colon follows the condition.

```
if (pop >= 22_000_000) and (gvt_type == "monarchy"):
    process_data()      # A hypothetical function
    record_results()    # Another hypothetical function
    ...
the_next_statement()
```

A conditional statement can also have two alternative bodies. If the condition is true, the first body is executed; otherwise (``else``), the second is executed.

```
if (pop >= 22_000_000) and (gvt_type == "monarchy"):
    process_data_set() # A hypothetical function
    record_results()   # Another hypothetical function
    ...
else:
    discard_data()     # One more hypothetical function
    ...
the_next_statement()
```

While it might seem inefficient to write two code blocks knowing that only one will be executed, remember that you will not know which one until the code runs.

11.1.1 Checking Membership

Consider the Boolean operator ``in`` as an example of a condition checking. This operator verifies whether an item exists in a container, such as a list, tuple, or set of dictionary keys. Its counterpart, ``not in``, checks the inverse condition.

```
students = ["Alice", "Bob", "Chuck"]
print("Alice" in students)
#> True
print("Abby" not in students)
#> True
```

The ``in`` operator can verify row/column membership in a dataframe. By default, it searches for column labels. To find a row label, examine the index membership.

```
df = df_sid.set_index("Name")
print(df)
#>          SID
#> Name
```

```
#> Bob      1234331
#> Chuck    5567615
#> Dinah    0019328
print("Dinah" in df)
#> False
print("SID" in df)
#> True
print("Dinah" in df.index)
#> True
```

Carrying out preemptive membership checks ensures that an item you plan to use exists within the container, preventing a potential failure when accessing it. In the following sections, you will explore this as one of the error prevention strategies.

11.2 Handling Programming Errors

Programming errors fall into three primary categories:

Syntax: Syntax errors occur when the grammar of your program is incorrect.

Discrepancies like unmatched quotation marks, inconsistent indentation, typographical errors in variable names, and missing semicolons, among others, can lead to lengthy error messages like those in Listing 9.1.1 on p. 171. While these messages might seem intimidating, they draw your attention to the error and assist you in resolving it. Syntax errors are arguably the least damaging as they are well-understood and easily rectifiable.

Logical: Logical errors are arguably the most problematic. A program with a logical error correctly solves an incorrect problem. For instance, you might mistakenly swap the function `Series.median()` with `Series.mean()`. They differ by only two characters and often yield similar results ([Section 11.4](#) on p. 226). Such errors may go unnoticed until it becomes costly or time-consuming to rectify, such as when a research paper has been submitted for peer review. The most effective way to evade logical errors is to continually scrutinize your results and methods, never take them as a given, and involve your colleagues in the review process.

Run-time: Run-time errors occupy a middle ground between syntax and logical errors. They result from unforeseen assumptions about your data, both the raw data extracted from the dataframes and series and the secondary, processed data. Let's talk more about them.

When dealing with potentially unreliable data in your code, you have two strategic options: preemptive checking or reactive recovery.

11.2.1 Preemptive Checking (Pessimistic Approach)

Here, you would examine the validity of your data or arguments before proceeding with your operation. Quality checks can involve various data validation or sanitization routines to ensure the data fits within expected arguments. You can confidently proceed with the operation if the data passes these checks. If it fails, you might halt the execution, return an error, or take another action. This approach is considered “pessimistic” because it operates under the assumption that data may be flawed or problematic.

The pessimistic approach is based on the conditional `if` statement. You can divide by a number `if` the number is not zero. You can get the index of a list item `if` the item is in the list. You can open a file for reading `if` the file exists and is readable. The general “pessimistic” framework is akin to Listing 11.1 on p. 218: define a suitable safety condition, evaluate it, and proceed if safe.

11.2.2 Reactive Recovery (Optimistic Approach)

The optimistic approach involves:

1. Proceeding with the operation.
2. Assuming the data is reliable.
3. Only taking action if and when a failure occurs.

If your operation results in an error due to problematic data, use recovery procedures to handle this situation. The optimistic approach hopes that your data will be good most of the time and errors will be the exception rather than the rule.

Technically, an exception is a condition resulting from a failure in the code, as illustrated in the following snippet, which attempts to open a nonexistent file:

```
datafile = "example.csv" # Must be "data/example.csv"!
with open(datafile, "r") as infile:
    data = infile.read()

#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> FileNotFoundError: [Errno 2] No such file or directory:
#>                               'example.csv'
```

The error message reports an exception that has been *raised*, but it is too late—the program has already crashed. You need another code fragment that *handles* the exception that was raised. For instance, with a missing file, the handler could at least inform the user about the issue in a user-friendly manner. However, it could also attempt to open a different file or prompt the user to re-enter the filename.

An exception-handling statement consists of the `try` block, one or more `except` blocks (one per each handled exception type), and an optional `finally` block that is out of this book's scope. The keywords `try`, `except`, and `finally` must be followed by a semicolon. The body of each block must be correctly indented.

The following three examples illustrate varying degrees of recovery attempts. The first example showcases the most generic recovery: all exceptions are quietly disregarded by executing a “do-nothing” `pass` statement. This recovery strategy is the least effective as it ignores the error, transforming a run-time error into a costly logical error.

```
try:
    with open(datafile, "r") as infile:
        data = infile.read()
except:
    pass
```

A more refined approach involves differentiating between file-related errors and others, providing clear and user-friendly error messages. This context offers a controlled version of an unhandled exception.

```
try:
    with open(datafile, "r") as infile:
        data = infile.read()
except FileNotFoundError:
    print(f"File not available: {datafile}")
except:
    print("Unspecified error")
#> File not available: example.csv
```

Finally, if you know other potential locations for the required file, you could design your program to fall back on these alternate data sources. The second `with` block should also be protected with an exception handler (not shown in this example).

```
try:
    with open(datafile, "r") as infile:
        data = infile.read()
except FileNotFoundError:
    with open("data/" + datafile, "r") as infile:
        data = infile.read()
except:
    print("Unspecified error")
```

In the above example, if the file is not found in the initially specified location, the program attempts to open the file from a different directory. This way, you build resilience into your code by planning for potential failures and ensuring your program can recover effectively.

The best approach to error handling depends on your situation’s specifics, such as the reliability of your data sources, the costs of encountering errors during operation, and the feasibility of validating data upfront or handling exceptions during run-time.

11.3 Counting, Frequency Distribution, and Visualization

Equipped with the safety instructions, let’s proceed with data exploration task.

The initial step in descriptive data analysis often involves counting data: calculating frequency distributions for each variable, by determining how frequently each value occurs (refer to [Section 3.1](#) on p. 34).

11.3.1 Counting with Counter

Counting can be performed with or without Pandas. The `Counter` data type from the `collections` module offers a quick and straightforward interface for counting. It enables you to count items in a list or tuple and stores the results in an object that behaves like a dictionary. The example below demonstrates its use (the sample data comes from Listing 10.1.1 on p. 191).

```
from collections import Counter
counts = Counter(words) # words defined above
print(counts)
#> Counter({'the ': 5, 'of ': 5, 'NLP': 3, 'is ': 3, 'as ': 3,
#> 'and': 3, 'language ': 3, 'a ': 2, 'at ': 1, 'core ': 1,
#> 'GPT-4': 1, ...})
```

The output reveals that “the” appears five times, “of” appears five times, “NLP” appears three times, etc. Have you noticed that of the top ten most frequently occurring words, seven (“the,” “of,” “is,” “as,” “and,” “a,” and “at”) are stopwords?

You can also get the count of a specific item:

```
print(counts["language"])
#> 3
```

The most common items and their counts can be retrieved with the `Counter.most_common()` function, which returns a list of tuples:

```
print(counts.most_common(2))
#> [('the ', 5), ('of ', 3)]
# What is the second most common word?
```

```
print(counts.most_common(2)[1][0])
#> of
```

As a bonus, you can convert the `counts` to a Pandas series and smoothly integrate them into the Pandas ecosystem—for instance, arrange the words in the descending order of their frequency.

```
counts_ser = pd.Series(counts).sort_values(ascending=False)
print(counts_ser.head())
#> the      5
#> of       5
#> and      3
#> language 3
#> is       3
#> dtype: int64
print(counts_ser["language"])
#> 3
```

11.3.2 Counting with Pandas

Alternatively, you can do counting directly in Pandas. When counting, we distinguish between absolute frequencies (the counts themselves) and relative frequencies (the counts relative to the total number of observations). The function `Series.value_counts()` was introduced in [Subsection 10.3.3](#) on p. 203, and this is all you need for determining absolute frequencies!

Using the example from [Section 10.4](#) on p. 210, you can calculate the absolute frequencies of regions in the dataset in the file “data/cc-region.csv”:

```
cc = pd.read_csv("data/cc-region.csv", index_col="Country Code")
abs_counts = cc.Region.value_counts().rename("Abs")
print(abs_counts)
#> Europe & Central Asia      59
#> Sub-Saharan Africa        49
#> Latin America & Caribbean  42
#> East Asia & Pacific        38
#> Middle East & North Africa  22
#> South Asia                8
#> North America             3
#> Name: Abs, dtype: int64
```

To obtain relative frequencies, divide the column of absolute frequencies by the column total, multiply by 100, and give the new series a proper name:

```
rel_counts = 100 * (abs_counts / abs_counts.sum()).rename("Rel")
```

You can now combine both series into a dataframe. [Subsection 10.4.1](#) on p. 211 explains how to concatenate Pandas series with `pandas.concat()`. The

keyword ``axis`` establishes the concatenation direction (0—vertical, stacked; 1—horizontal, side by side). Though not generally recommended, you can even calculate the totals of both columns and append them to the bottom of the dataframe.

```
freqs = pd.concat([abs_counts, rel_counts], axis=1)
totals = freqs.sum().astype(int).rename("Total")
freqs_tot = freqs.append(totals )
print(freqs_tot)
```

	<i>Abs</i>	<i>Rel</i>
<i>#> Europe & Central Asia</i>	59	26.696833
<i>#> Sub-Saharan Africa</i>	49	22.171946
<i>#> Latin America & Caribbean</i>	42	19.004525
<i>#> East Asia & Pacific</i>	38	17.194570
<i>#> Middle East & North Africa</i>	22	9.954751
<i>#> South Asia</i>	8	3.619910
<i>#> North America</i>	3	1.357466
<i>#> Total</i>	221	100.000000

11.3.3 Histograms and Bar Charts

You can visualize absolute and relative frequencies using histograms (for continuous data) and bar charts (for categorical data). If this is your first encounter with Matplotlib, a Python data visualization library, you must import it, as explained later.

As an example of a histogram, let's apply the ``Series.hist()`` function to generosity. Although the function accepts numerous arguments, it usually works well with default values. However, the default bin number is 10, leading to rough charts, so consider increasing it.

The histogram is rendered on the "figure": Matplotlib's drawing canvas. Depending on your IDE settings, the figure might appear immediately or only after you invoke ``plt.show()``. You can use these and many other functions to control the figure's appearance:

- ``plt.xlabel(label)``: modify the X-axis label;
- ``plt.ylabel(label)``: modify the Y-axis label;
- ``plt.title(title)``: add a title;
- ``plt.grid()``: toggle the grid lines;
- ``plt.xlim(min, max)``: control the X-axis extents;
- ``plt.ylim(min, max)``: control the Y-axis extents;
- ``plt.xscale("log")``: set the X-axis scale to logarithmic;
- ``plt.yscale("log")``: set the Y-axis scale to logarithmic.

Before displaying the final figure, save it to a file for future use.

```
import matplotlib.pyplot as plt

generosity.hist(bins=10, color="gray")
plt.xlabel("Generosity")
plt.ylabel("Frequency")
plt.title("World Happiness Report/2021")
plt.tight_layout() # Do you want less white space?
plt.savefig("figures/generosity.pdf")
plt.show()
```

Figure 11.1 shows the output of the code fragment.

Histograms categorize items into predetermined numerical bins. They are not recommended for categorical or labeled data, which are better represented with bar charts introduced in [Subsection 3.1.1](#) on p. 36. Matplotlib provides the capability to draw horizontal and vertical bar charts using the functions `Series.plot.barh()` and `Series.plot.bar()`, respectively, available through the `Series.plot` accessor.

The number of bins (or bars) in a bar chart is automatically determined, equating to the number of categories. The width of the bars in the following

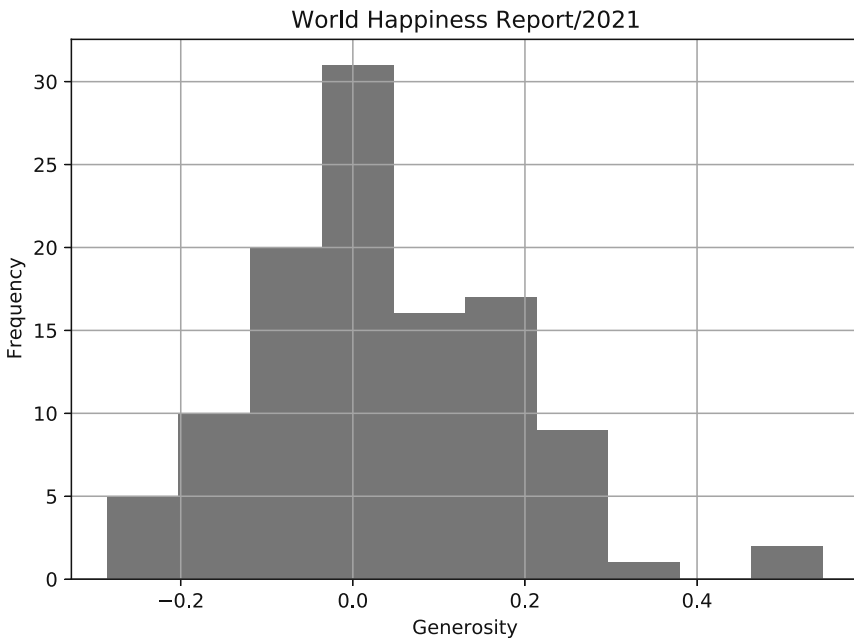


FIGURE 11.1

Histogram illustrating the distribution of generosity.

examples is 80% of the inter-bar distance. The charts are neatly organized using `plt.tight_layout()` to ensure the bar labels do not extend beyond the canvas boundary.

```
abs_counts[::-1].plot.barh(width=0.8, color="gray")
plt.xlabel("Frequency")
plt.tight_layout()
plt.savefig("figures/barh2.pdf")
plt.show()
```

In the case of the vertical bar chart, the bar labels are rotated 30° around their top-right corners to prevent them from overlapping.

```
abs_counts.plot.bar(width=0.8, color="gray")
plt.ylabel("Frequency")
plt.xticks(rotation=30, ha="right", va="top",
            rotation_mode="anchor")
plt.tight_layout()
plt.savefig("figures/bar2.pdf")
plt.show()
```

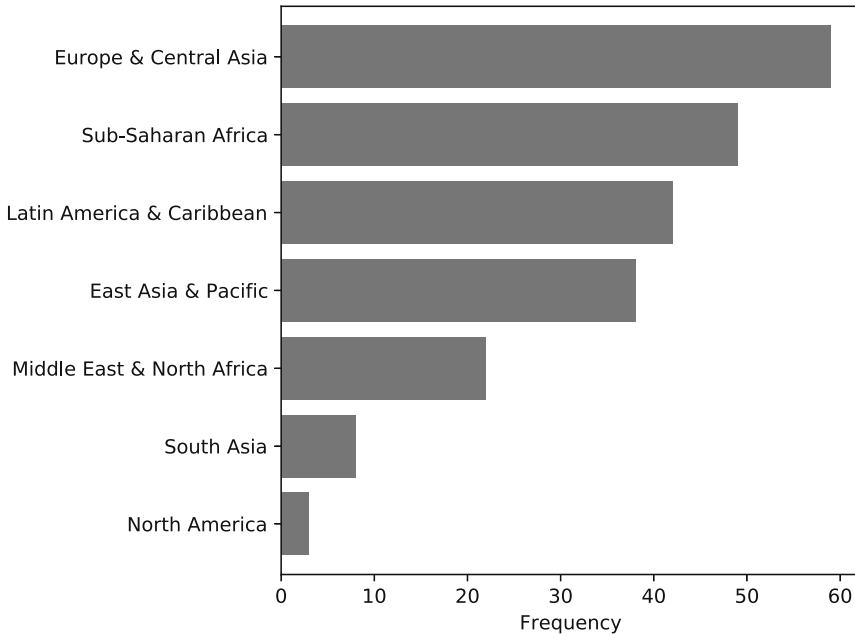
Figures 11.2 and 11.3 show the output of the code fragments.

11.4 Descriptive Statistics Functions

The powerful `Series.describe()` function reports basic statistics of central tendency (mean, standard deviation, minimum, maximum, and quartiles, [Subsection 3.1.2](#) on p. 41) of the variable contained within a series. Here is the familiar `generosity` series from the `happy21` dataframe:

```
generosity.describe()
#> count    111.000000
#> mean      0.033183
#> std       0.148975
#> min      -0.285531
#> 25%      -0.073578
#> 50%       0.020692
#> 75%       0.133323
#> max       0.545828
#> Name: Generosity, dtype: float64
```

Each statistic can be calculated separately from the others if needed—[Table 11.1](#) explains how. The same functions work for a series when they return just one number, not another series.

**FIGURE 11.2**

Bar chart illustrating the distribution of countries by region (vertical).

In a slightly different spirit, the function `DataFrame.mode()` returns a collection of series of all modes of all numeric variables in decreasing order. The principal mode (“the Mode”) is in each series’ zeroth position.

```
print(generosity.mode()[0])
#> -0.285530955
```

11.4.1 Data Imputation Revisited

Knowing the mean value of a continuous series or the mode of a categorical series allows you to perform missing data imputation with better accuracy than imputing a constant, as in Listing 10.3.6 on p. 209.

```
missing_gpa = df_with_zack["GPA"].isnull()
mean_gpa = df_with_zack["GPA"].mean()
df_with_zack.loc[missing_gpa, "GPA"] = mean_gpa

missing_rank = df_with_zack["Rank"].isnull()
mode_rank = df_with_zack["Rank"].mode()[0]
df_with_zack.loc[missing_rank, "Rank"] = mode_rank
```

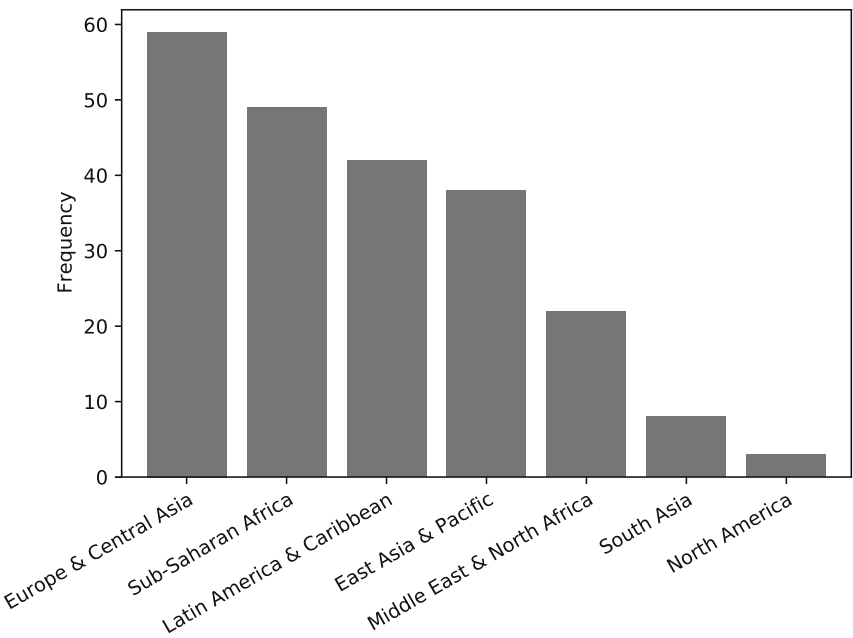



FIGURE 11.3
Bar chart illustrating the distribution of countries by region (horizontal).

TABLE 11.1
Functions for calculating statistics of central tendency

Function	Result
<code>`DataFrame.min()`</code>	The smallest values of all numeric variables
<code>`DataFrame.max()`</code>	The largest values of all numeric variables
<code>`DataFrame.mean()`</code>	The means of all numeric variables
<code>`DataFrame.median()`</code>	The medians of all numeric variables
<code>`DataFrame.quantile(q)`</code>	The values of the <code>`q`</code> 's quantiles of all numeric variables
<code>`DataFrame.std()`</code>	The standard deviations of all numeric variables
<code>`DataFrame.var()`</code>	The variances of all numeric variables

Sadly, imputing the mean or the median still biases the data statistics, especially the standard deviation.

11.5 Identifying and Interpreting Outliers

[Subsection 3.1.4](#) on p. 47 defines an outlier as an observation that significantly diverges from the mean value of a given variable. This divergence is typically set at three or more standard deviations ($\pm 3\sigma$). In the example, let's identify outliers within the annual observations of negative affect in India. However, the sample does not contain any observation that deviates from the mean by the standard threshold of three standard deviations. Given this situation, let's take the liberty to reduce the threshold to $\pm 1.5\sigma$ for this analysis.

```
# Prepare a sample
year_neg = happy.loc[happy["Country name"] == "India"]\
               .set_index("year")["Negative affect"]

# Calculate outliers
THRESHOLD = 1.5
deviation = (year_neg - year_neg.mean()).abs()
is_outer = deviation > THRESHOLD * year_neg.std()
outliers = year_neg[is_outer]

print(outliers)
#> year
#> 2006      0.198602
#> 2019      0.466336
#> 2021      0.429138
#> Name: Negative affect, dtype: float64
```

[Figure 11.4](#) illustrates the trend of negative affect. Dots represent regular values, whereas the three outliers we identified are denoted with crosses. It is clear that what we classified as “outliers” earlier do not truly fit this description. The negative affect exhibits a positive linear trend, which shifts its frequency distribution away from the norm. Given that the outlier detection method mentioned in [Subsection 3.1.4](#) is tailored towards variables with a generally normal distribution, these three identified observations should not be considered outliers.

11.6 Utilizing Correlation to Analyze Variable Relationships

Linear Pearson correlation, also known simply as a correlation, elevates data analysis to the next level, transitioning from examining individual variables

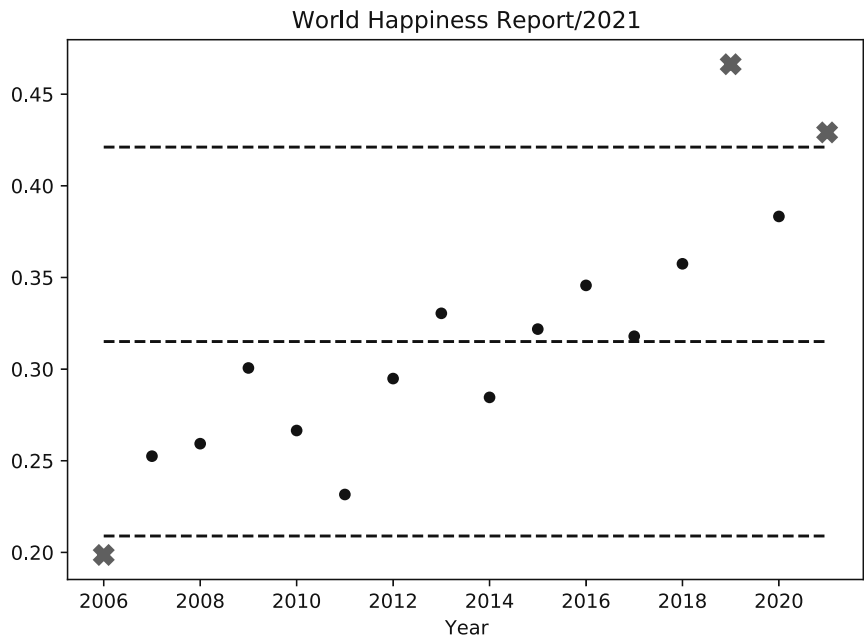


FIGURE 11.4
Negative affect in India: regular values (dots) and outliers (crosses).
Threshold= $\pm 1.5\sigma$.

to exploring relationships between them. This process is a crucial component of exploratory analysis, as discussed in [Section 3.2](#) on p. 47.

The correlation r_{xy} between two continuous variables x and y with respective mean values $\bar{x}$ and $\bar{y}$, and standard deviations σ_x and σ_y is defined as:

$$r_{xy} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sigma_x \sigma_y},$$

where x_i and y_i are the i th observations of x and y .

Pandas offers a simpler approach to this process with the `DataFrame.corr()` function, which computes the correlations between all pairs of numeric variables in the dataframe, including each variable’s correlation with itself (which, by definition, equals 1).

For a dataframe with 11 numeric columns (e.g., `happy_full`), the output of `happy_full.corr()` is an 11×11 dataframe with 11 ones, 55 unique correlations, and 55 duplicates (because $r_{xy} = r_{yx}$). You can extract the individual correlations using the `.loc` accessor.

```
corr_mtx = happy_full21.corr()
print(corr_mtx.loc["Social support", "Log GDP per capita"])
#> 0.7535397932341442
```

To identify the most positively and negatively correlated variables in your dataframe, filter out trivial identities and duplicates before sorting the remaining correlations.

[Subsection 10.4.3](#) on p. 213 proposed a way to “lengthen” a dataframe by converting row and column indexes into a two-level MultiIndex through a stacking operation, then resetting it. This technique can be applied to the correlation table, with the new columns being renamed to reflect their contents better.

```
corrs = corr_mtx.stack().reset_index()\
    .rename(columns={"level_0": "var0",
                    "level_1": "var1",
                    0: "r"})
```

The dataframe `corrs` is a long-form dataframe with 121 rows and three columns representing variable pairs and their corresponding correlations. Rows, where the two variables are identical, are instances of self-correlation and should be removed.

You should also remove symmetric correlations, that is, correlations where two variables are reversed in their pairing. For instance, if you have calculated the correlation for the pair (“var0”, “var1”), you do not need also to calculate the correlation for (“var1”, “var0”). You can enforce this by only considering pairs where the first variable comes alphabetically after the second.

Finally, focus on strong correlations with an absolute value larger than 0.7. There are six such correlations in the dataset. They are all positive and pertain to four variables: “Social support,” “Life Ladder,” “Log GDP per capita,” and “Healthy life expectancy at birth.”

```
corrs = corrs[corrs["var0"] > corrs["var1"]].sort_values("r")
print(corrs[corrs["r"].abs() >= 0.7])
#>
#>      var0                                var1      r
#> 23  Social support  Healthy life expectancy... 0.718436
#> 3   Life Ladder    Healthy life expectancy... 0.726250
#> 21  Social support                                Log GDP per capita 0.753540
#> 10  Log GDP per capita                                Life Ladder 0.792734
#> 20  Social support                                Life Ladder 0.818278
#> 13  Log GDP per capita  Healthy life expectancy... 0.855115
```

The correlation between “Log GDP per capita” and “Healthy life expectancy at birth” is the strongest. According to the scatter plot in [Figure 11.5](#), the relationship between the two variables is solid, though soiled by a few outliers.

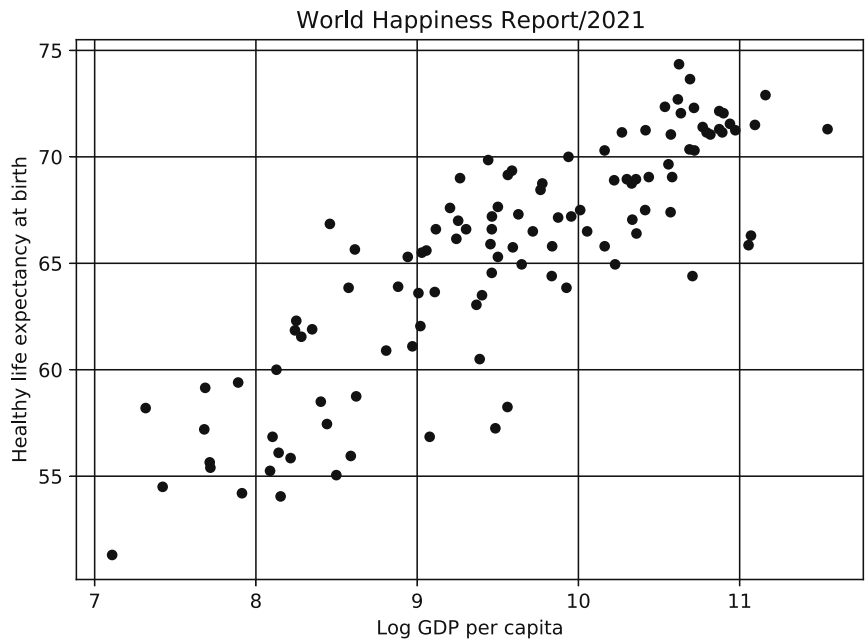


FIGURE 11.5
Healthy life expectancy at birth *vs.* log GDP per capita in 2021. The correlation between the variables is 0.855.

11.7 What’s Next

Statistical analysis is often mundane, requiring endless repetitions of simpler operations, such as hypothesis testing. In the next chapter, you will learn about hypothesis-testing techniques and how to automate them with Python loops.

11.8 Programming Exercises

Exercise 10. Variables `pop1`` and `pop2`` denote the population of Oneland and Twoland. Write one Python expression that is true if and only if Oneland has at most 10% of the population of Twoland or Twoland has at most 10% of the population of Oneland.

Exercise 11. Write a code fragment that reports the percentage of rows in the ``happy`` dataframe with generosity above the mean and above the median (two numbers).

Exercise 12. Dataframe ``cc`` (“data/cc-region.csv”) contains column “TableName” representing country names. Count the frequencies of the first letters of the country names (as “A” in “Australia”) and plot them as a horizontal bar chart.

Exercise 13. Report the countries from the ``happy`` dataframe that are outliers with respect to “Social support.” Use the “three sigmas” rule. Use the function ``Series.unique()`` to eliminate duplicates from the result.

Loops and Hypothesis Testing

Statistical analysis often involves tasks, such as non-parametric hypothesis testing, that require the repeated execution of certain operations. This repetition can be time-consuming and labor-intensive if done manually. Here is where the real strength of a computer shines—unlike a calculator, a computer can be programmed to automate these repetitive tasks. In programming parlance, such repeated execution is known as *looping*.

12.1 Repeating Operations with Loops

Familiarity with Python’s looping structures is critical to avoid the drudgery of manually re-executing your code. Python provides two types of loop constructs: the “while” loop and the “for” loop, which are common features in modern programming languages.

The “while” loop enables an operation to be repeated while (hence the name) a particular condition remains true. Conversely, the loop will cease once the condition becomes false. Each complete execution of a loop is called an iteration.

The “for” loop is, in truth, a “for-each” loop. It *iterates* (steps) through a container by applying the same operation to each item in the container, optionally filling another container with the transformed items.

12.1.1 “While” Loops

Let’s look at a simple “while” loop example: Suppose we have a discrete uniform random number distribution ranging from 0 to 9. Our task is to determine how many draws it would take to yield a specific number—say, seven.

The solution involves three steps that will be continually repeated until the desired outcome is achieved:

1. Draw a random number by calling ``random.randint()`.`
2. Compare this number to the target, seven.
3. If the numbers do not match, add one to the counter tracking the number of attempts, and then repeat the process.

This situation is ideally suited for a “while” loop. The structure of a “while” loop is similar to a conditional statement ([Section 11.1](#) on p. 217): it starts with the keyword `while`, followed by the condition determining whether the loop should continue, and finally, the appropriately indented block of code to be repeatedly executed.

```
import random

counter = 1 # The counter. We need at least one attempt!

number = random.randint(10) # Step 1
while number != 7:           # Step 2
    counter += 1             # Step 3
    number = random.randint(10) # Step 1 repeated
print(counter)
```

Note that in this code, the first step is programmed twice: before the loop for the initial draw and within the loop for all subsequent attempts. This redundancy is necessitated by the loop structure: the condition is always checked before any other operation is performed. However, in our case, we need to generate a random number before checking the condition. To resolve this, we preemptively draw the first number and re-draw it within the loop. The loop stops when our drawn number is equal to seven.

As another example, consider using a “while” loop to validate a large CSV file (see code listing on p. 195). Assuming that the file data does not contain commas (i.e., commas only separate columns), we aim to count the number of columns in each row and raise an error if an inconsistency is found. Since the file is large, we must read it line by line rather than using `TextIOWrapper.read()` until an empty string signifies the end of the file.

```
with open("data/example.csv", "r") as infile:
    line = infile.readline()
    count = len(line.split(","))

    while line != "": # Same as `while line:`
        line = infile.readline()
        if count != len(line.split(",")):
            print(f"Malformatted line: '{line}'")
            raise Exception # raise an exception
```

In this code snippet:

- The variable `count` holds the number of columns, determined by splitting the initially read line by commas. Then, we read the next line, split it, and compare the number of columns to the count stored in `count`.
- If the counts differ, the line is malformed. In this case, we print an error message and use the `raise` keyword to trigger an exception. [Subsection 11.2.2](#)

on p. 220 explains how to handle the raised exception. If you prefer to inspect the rest of the file before failing (often considered best practice), you should allow the loop to continue after printing the error instead of raising an exception.

If you want a gentler way to stop the loop prematurely, use the ``break`` statement, as in the following code:

```
...
    if count != line.split(","):
        print(f"Malformatted line: '{line}'")
        break
...
```

This ``break`` statement immediately skips to the next piece of code that follows the loop in your program. It will not crash the program and does not necessitate exception handling.

12.1.2 “For” Loops

A “for” loop, unlike a “while” loop, iterates over a container, such as a string, list, or tuple, performing the same operation on each item. A “for” loop involves the ``for`` keyword, followed by a loop variable, the ``in`` keyword, the container, and then the properly indented block of code to be executed. The loop variable is a placeholder that will take on the value of each item in the container as the loop iterates.

Often, if the container represents a plural or collective noun (e.g., “letters,” “people,” “residents”), the loop variable may be named after the singular form of that noun (“letter,” “person,” “resident”).

At the start of the loop’s first iteration, the loop variable is assigned the value of the first item in the container. If the container is empty, the loop is terminated. If not, the block of code is executed. Subsequently, the loop variable is assigned the value of the next item in the container, and the process repeats. The number of iterations is never greater than the number of items in the container; a loop can be prematurely terminated via a ``break`` statement or by raising an exception.

The following examples illustrate various use cases for “for” loops.

```
vowels = 0
for letter in "Hello, world!":
    if letter in "aeouiAEQUI":
        vowels += 1
print(vowels)
#> 3
```

This code snippet counts the number of vowels in a string. You can intuitively interpret a “for” loop by mentally adding the word “each” before

the loop variable declaration and the word “that” before each usage of that variable in the loop body. Please note that this is not a valid code fragment but a way to understand the concept.

```
vowels = 0
for EACH letter in "Hello, world!":
    if THAT letter in "aeouiAEoui":
        vowels += 1
```

Note that the `in` keyword serves a dual purpose in the example. In the header, it specifies the source of the items. In the loop body, it checks the membership of an item in another container ([Subsection 11.1.1](#) on p. 218).

A “for” loop often utilizes an accumulator variable. Much like an electrical accumulator gathers electric charge, an accumulator variable collects items over iterations. Typically, an accumulator may be a number (used for quantities), a string (used for compiling characters), or a list (used for collating other types of items). An accumulating “for” loop begins with an empty/zero accumulator, and new items are appended or added to it as the loop progresses.

In the following example, a sentence is split into “words” by white spaces. The words are then separated into two lists: “shorter” and “longer,” based on their lengths. We consider any word with more than five characters a “longer” word.

```
shorter = [] # An accumulator
longer = [] # Another accumulator
sent = """All human beings are born free and equal in dignity
and rights. They are endowed with reason and conscience and
should act towards one another in a spirit of brotherhood."""

for word in sent.split():
    if len(word) > 5:
        longer.append(word)
    else:
        shorter.append(word)

print(";".join(shorter))
#> All;human;are;born;free;and;equal;in;and;They;...;one;in;a;of
print(";".join(longer))
#> beings;dignity;rights.;endowed;reason;...;brotherhood.
```

In this case, variables `shorter` and `longer` are accumulators. They begin as empty lists and get populated with “shorter” and “longer” words as the loop iterates over the sentence.

12.1.3 List Comprehension

In data analysis, it is common to see a “for” loop with an accumulator comprising a single statement. This statement usually checks a condition (optionally),

transforms the current value of the loop variable (optionally), and then appends it to the accumulator (again, optionally). Such a loop can be effectively converted into a list comprehension.

List comprehensions provide a concise way to create lists based on existing lists or other containers. They make your code more readable and sometimes more efficient.

A list comprehension can be considered a compact version of a “for” loop combined with an “if” statement. The general form of a list comprehension with an assignment statement is:

```
result = [expression for var in container if condition]
```

In this form:

- The result of a list comprehension is another list, possibly empty.
- ``var`` is the variable used for each item in the container.
- ``expression`` is the operation you wish to apply to each item in the container. The expression can be the variable itself if no transformation is needed.
- ``condition`` is an optional inclusion criterion that must be satisfied for an item to be included in the new list. If absent, all items are included.

The expression is fully equivalent to the following statement but is more compact.

```
result = []
for var in container:
    if condition == True: # Optional
        result.append(expression)
```

This simplest list comprehension converts a list into an identical list, essentially copying the list:

```
list2 = [item for item in list1] # Same as list2 = list1.copy()
```

A more practical list comprehension can separate “shorter” and “longer” words from a given sentence (referring to the example on p. 237):

```
words = sent.split()
shorter = [word for word in words if len(word) <= 5]
longer = [word for word in words if len(word) > 5]
```

One limitation of list comprehension is that it can only generate a single result. If you need multiple accumulations and want to avoid traversing a lengthy container several times, a standard “for” loop would be more suitable.

As a final example, let’s use list comprehension to calculate the average word length in the same sentence. This task involves generating a list of word lengths, summing them, and dividing the sum by the total number of words:

```
words = sent.split()
lengths = [len(w) for w in words] # No condition!
mean_length = sum(lengths) / len(lengths)
print(mean_length)
#> 4.7
```

This list comprehension does not use a condition as it counts all words, but it employs a non-trivial expression, `len(word)`. Incidentally, the middle two lines of this code work for any container holding items that can be measured (i.e., items for which the function `len(x)` is valid). Once you learn to create custom functions (in [Section 13.1](#) on p. 249), you might consider “functionalizing” these two lines for reuse.

With an understanding of loops and list comprehensions, you can automate any mundane activity and concentrate on solving statistical problems.

12.2 Parametric Hypothesis Testing

Much of Python’s statistical analysis tools come from the module, which is somewhat ironically called SciPy (“Scientific Python”) as if other Python modules could not be used in science. SciPy contains submodules for optimization, linear algebra, integration, interpolation, signal and image processing, etc. The submodule you will need in this chapter is `scipy.stats` (for “statistics”).

```
import scipy.stats as sts
```

Module `statsmodel` is another Python module that provides statistical tools, such as descriptive statistics, non-parametric tests, power calculation, ANOVA, and statistical plots. `statsmodel` is a large module; you will only need some of its content at the same time. We will import it piecewise as necessary.

12.2.1 Shapiro–Wilk Test

The normality of a statistical variable is a frequent assumption in statistics. The Shapiro–Wilk test introduced in [Subsection 4.3.4](#) on p. 65 tests the null hypothesis that the data `X` (a list or a one-dimensional NumPy array) was drawn from a normal distribution. The function `sts.shapiro(X)` performs this test and returns a named tuple, the elements of which can be accessed using dot notation. If you assign the tuple to a variable `result`, `result.statistic` holds the test statistic and `result.pvalue` the associated p-value.

If the p-value exceeds 0.05, the data can be treated as normal; otherwise, the data significantly deviates from a normal distribution. The following

examples illustrate this with synthetic data (randomly drawn from a normal distribution):

```
import numpy.random as rnd # For random number generators
import pandas as pd
synthetic_data = rnd.normal(size=500)
result = sts.shapiro(synthetic_data)
print(result)
#> ShapiroResult(statistic=0.995905518532, pvalue=0.220757648349)
if result.pvalue > 0.05:
    print("The data appears to be normal")
else:
    print("the data deviate from a normal distribution")
#> The data appears to be normal
```

Let's perform the same test on empirical data, specifically the "Life Ladder" scores for Sub-Saharan Africa (p. 65):

```
the_region = "Sub-Saharan Africa"
subset = happy_full21["Region"] == the_region
subsaharan_data = happy_full21.loc[subset, "Life Ladder"]
result = sts.shapiro(subsaharan_data)
print(result)
#> ShapiroResult(statistic=0.9758, pvalue=0.8248)
```

For a more tangible sense of the distribution, you can plot a histogram of the empirical dataset, following the instructions in [Subsection 11.3.3](#) on p. 224.

In the counter-example provided, the Shapiro–Wilk test is applied to a dataset known not to be normally distributed—a uniform distribution.

```
uniform_data = rnd.uniform(size=500)
result = sts.shapiro(synthetic_data)
print(result)
#> ShapiroResult(statistic=0.9576, pvalue=0)
```

Here, the p-value is extremely low (approximately 8.45×10^{-11}), far less than the conventional threshold of 0.05. This results in rejecting the null hypothesis, implying the data significantly deviates from a normal distribution. As such, this result is consistent with our expectations for a dataset drawn from a uniform distribution, which should not exhibit the bell-curve shape of a normal distribution.

12.2.2 Z-Test

The Z-test compares a sample mean with the population mean and tests whether the difference between them is statistically significant. In Python,

it is implemented as an almost-namesake function `ztest()` from the module `statsmodels.stats.weightstats`¹. The function returns a two-item tuple (not a named tuple). The first item is the Z-score; the second is the p-value.

```
from statsmodels.stats.weightstats import ztest
result = ztest(happy_full121["Life Ladder"], subsaharan_data)
print(result)
#> (4.6561, 0)
```

In this context, the Z-score gives you an idea of how many standard deviations the sample is from the mean, while the p-value tells you the probability that you would see such an extreme Z-score if the null hypothesis were true. The p-value is 0, indicating that the Sub-Saharan sample differs from the rest of the world regarding the Ladder score.

12.2.3 T-Test

The t-test is similar to the Z-test but does not require knowledge of population statistics, which is often challenging to acquire. This test compares two samples, with the null hypothesis positing that both are drawn from an identical population.

The independent samples t-test compares the samples from two unrelated groups (e.g., the Ladder scores for “East Asia & Pacific” and “Middle East & North Africa”). The paired t-test assumes that the samples are related (e.g., the Ladder scores for the countries from the same region in different years).

Both `statsmodels` and `scipy.stats` provide functions for these t-tests. They accept the same arguments and yield the same results: a named tuple with `statistic` and `pvalue`. Here is how you might utilize the `scipy.stats` implementation. First, let’s examine two independent samples:

```
the_region1 = "East Asia & Pacific"
the_region2 = "Middle East & North Africa"
d1 = happy_full121[happy_full121["Region"] == the_region1]
d2 = happy_full121[happy_full121["Region"] == the_region2]
result = sts.ttest_ind(d1["Life Ladder"], d2["Life Ladder"])
print(result)
#> Ttest_indResult(statistic=1.3817, pvalue=0.1788)
```

The high p-value of 0.1788 signifies that the null hypothesis cannot be rejected, and the Ladder scores do not differ significantly between the two regions.

The paired (“related”) t-test analyzes changes in the same region (“East Asia & Pacific”) from 2011 to 2021. However, some countries are absent in the incomplete dataset in 2011 (Myanmar) or 2021 (Malaysia). With the abundance of data in the adjacent years, you can impute (patch) the missing values

¹Module `scipy.stats` has a similar function, `zscore()`, for calculating the Z-score.

by interpolating (where possible), forward filling, and then backward filling in the horizontal (temporal) dimension (consult [Subsection 10.3.6](#) on p. 208).

```
# We need the full dataset, not just 2021
the_region = "East Asia & Pacific"
eap_df = happy_full[happy_full['Region'] == the_region]
eap_df = eap_df.pivot(index="Country name", columns="year",
                      values="Life Ladder")

# Patch the missing values
eap_df = eap_df.interpolate(axis=1).bfill(axis=1).ffill(axis=1)
result = sts.ttest_rel(eap_df[2011], eap_df[2021])
print(result)
#> Ttest_relResult(statistic=-0.3122, pvalue=0.7589)
```

A p-value of 0.76 shows that the Ladder score in the “East Asia & Pacific” region remained statistically unchanged.

It is important to remember that both Z-test and t-test can be applied only to normally distributed data. Use the Shapiro–Wilk test to validate their applicability.

12.2.4 Analysis of Variance (ANOVA) Test

Analysis of Variance (ANOVA) is a collection of statistical models and corresponding estimation procedures used to analyze differences among group means.

One-way ANOVA

The one-way ANOVA is employed to investigate the relationship between a continuous dependent variable and a single categorical independent variable. For illustration, consider analyzing the Ladder score across three distinct regions: East Asia & Pacific, Middle East & North Africa, and Europe & Central Asia. The Ladder score is the dependent variable, while the specific regions are categories within the independent variable. The null hypothesis posits that the mean score across regions is equal.

There are two notable Python functions to perform the one-way ANOVA test: `f_oneway()` from `scipy.stats` and `anova_oneway()` from `statsmodels.stats.oneway`. Both methods presuppose independent observations within each category and a roughly normal distribution across categories. If these assumptions are violated, alternative statistical methods may be more suitable. Additionally, the `f_oneway()` method requires the variances within each category to be approximately equal, a concept known as homoscedasticity.

Here is a code example that illustrates how to perform a one-way ANOVA. Start by preparing the data as a tuple of region names and extracting Ladder scores for each region. Choose a tuple over a list because lists are mutable, and tuples are not. The tuple notation highlights that the collection will not

change while solving the problem. After executing the list comprehension, ``scores`` is a list of lists of scores.

```
the_regions = ("East Asia & Pacific",
               "Middle East & North Africa",
               "Europe & Central Asia")
scores = [happy_full21.loc[happy_full21["Region"]==reg,
                           "Life Ladder"].values
          for reg in the_regions]
```

Function ``sts.f_oneway()`` expects each group of scores to be passed as one argument. For three groups, you should pass three arguments: ``scores[0]``, ``scores[1]``, and ``scores[2]``. The Python unpacking operator ``*`` allows passing several arguments as a list, such that the first list item becomes the first argument, etc. Below, ``sts.f_oneway(*scores)`` is identical to ``sts.f_oneway(scores[0], scores[1], scores[2])`` but is more compact and does not depend on the number of items.

The p-value of 0.0009 produced by ``sts.f_oneway()`` leads to rejecting the null hypothesis, implying a significant relationship between the Ladder score and the region. Meanwhile, the other ANOVA function, ``anova_oneway()`` from ``statsmodels.stats.oneway``, is more skeptical. It returns a p-value of 0.0132, still significant but less extreme.

```
anova = sts.f_oneway(*scores)
print(anova)
#> F_onewayResult(statistic=7.7615, pvalue=0.0009)

from statsmodels.stats.oneway import anova_oneway
anova_oneway(scores).tuple
#> (5.2987, 0.0132)
```

The discrepancy is attributed to the heteroscedastic nature of the data, with the second group's standard deviation being notably larger. The correct p-value is reported by ``anova_oneway()``, but it can also mimic the result of ``sts.f_oneway()`` by forcing equal variances with the ``use_var="equal"`` option.

```
[x.std() for x in scores]
#> [0.7713646745315361, 1.4130118576091986, 0.7779813029601288]
>anova_oneway(scores, use_var="equal").tuple
#> (7.7615, 0.0009)
```

However, these equalized p-values are incorrect due to the underlying heteroscedasticity of the data.

Two-way ANOVA

The two-way ANOVA test extends the one-way ANOVA to allow an analysis of how two independent variables, either continuous or categorical, influence a

dependent variable. Built on Ordinary Least Squares (OLS) linear regression, two-way ANOVA is related to concepts detailed in [Chapter 13](#). After reading that chapter, revisiting this section for a deeper understanding is advisable.

Suppose you want to investigate how the region and income group influence the mean Ladder score. The following Python code leverages the `ols()` function from ``statsmodels.formula.api`` to create and fit an OLS regression model. The ``anova_lm()`` function from ``statsmodels.api.stats`` interprets the regression results.

The *formula* parameter, expressed as the string ``"Life_Ladder ~ C(Region) + C(IncomeGroup)"``, defines the dependent variable ``Life_Ladder`` and the two independent variables ``Region`` and ``IncomeGroup``. The ``C()`` function denotes these independent variables as categorical.

```
import statsmodels.api as sm
from statsmodels.formula.api import ols

happy_full21["Life_Ladder"] = happy_full21["Life Ladder"]
formula = "Life_Ladder ~ C(Region) + C(IncomeGroup)"
model = ols(formula, data=happy_full21).fit()
sm.stats.anova_lm(model, typ=2)
```

#>		sum_sq	df	F	PR(>F)
#>	C(Region)	13.476147	6.0	5.588006	4.491285e-05
#>	C(IncomeGroup)	33.355488	3.0	27.662306	2.541935e-13
#>	Residual	43.007227	107.0	NaN	NaN

The column labeled “PR(>F)” in the result, the p-value in disguise, is of particular interest. Extremely low p-values for both the region and income group lead to rejecting the null hypotheses. Therefore, assuming one of the independent variables remains constant, the other independent variable significantly impacts the Ladder score.

12.2.5 Chi-Squared Test

The chi-squared test, also known as the χ^2 test, has two primary applications: testing for independence between two categorical variables and serving as a goodness-of-fit test for a single categorical variable. The remainder of this subsection focuses on its role as a test for independence.

Consider the case of two categorical variables, “Income Group” (V_1) and “Corruption” (V_2). The second variable is not a part of the original dataset, but you can produce it by applying the ``pd.qcut()`` function (see p.204) to the existing continuous “Perceptions of corruption” variable. The variables have $N_1 = 4$ and $N_2 = 3$ levels, respectively.

```
c = happy_full21["Perceptions of corruption"]
# Categorizing
labels = ("low", "med", "high")
happy_full21["Corruption"] = pd.qcut(c, len(labels), labels)
```

Your next step is to perform cross-tabulation between these variables, resulting in an $N_1 \times N_2$ integer table known as a contingency table. Each cell in this table's i th row and j th column contains the count of samples at the i th level of V_1 and the j th level of V_2 . [Table 4.2](#) on p. 69 is an example of a contingency table, and the `pandas.crosstab()` function facilitates its calculation.

```
# Cross-tabulating
contingency = pd.crosstab(happy_full121["Corruption"],
                          happy_full121["IncomeGroup"])
result = sts.chi2_contingency(contingency)
print(result[0], result[1])
#> 22.1508 0.0011
```

Given the extremely low p-value in this example, the null hypothesis implying no association between V_1 and V_2 should be rejected. This observation suggests that statistically significant relationships exist between the “Income Group” and “Corruption.”

Note: The chi-squared test is unsuitable when the observed frequencies in each category of the contingency table are too low. A commonly accepted guideline is that all frequencies should be at least 5.

12.3 Non-Parametric Hypothesis Testing via Bootstrapping

Non-parametric hypothesis testing does not rely on specific assumptions about the data distribution. It is ideal when the data does not conform to normal distribution assumptions.

Bootstrapping, a non-parametric method, involves resampling data with replacement to approximate the sampling distribution of a statistic. This process entails randomly drawing observations from the original data set until the bootstrap sample equals the original dataset's size.

To explore bootstrapping in the spirit of the example from [Section 4.4](#) on p. 72, let us re-examine the hypothesis stating that the Ladder score in Sub-Saharan Africa differs from the rest of the world. A convenient method is to employ the function `sts.mannwhitneyu()`, which performs the Mann-Whitney U rank test on two independent samples, testing the null hypothesis that the distributions underlying the samples are identical.

```
sts.mannwhitneyu(subsaharan_data, happy_full121["Life Ladder"])
#> MannwhitneyUResult(statistic=527.5, pvalue=1.62719e-06)
```

The low p-value mandates the rejection of the null hypothesis.

The direct approach based on the Mann–Whitney test is usually all you need—unless you want to compare standard deviations, variances, or other statistics. In that case, you must code the bootstrapping algorithm from [Section 4.4](#) on p. 70. In particular, you will have to select a sample with replacement and repeat that process many (1,000 or more) times.

The Pandas function `Series.sample()` will help you accomplish the first task, with the option `replace=True` enabling sampling with a replacement. The subsequent task involves looping to create many samples and computing the means, which can be done using either a “while” loop with an explicit counter variable (try it; see Problem 14 on p. 248) or a “for” loop iterating through a list of 1,000 numbers generated by the function `range()`. The numbers on the list are irrelevant; they are used to fill up the container. We will call the loop variable `_` (an underscore) to emphasize its function as a placeholder.

The bootstrapping procedure starts by creating an empty list `sample_means` and then iteratively samples the population, calculates the sample mean, and appends it to the list. After 1,000 iterations, the list is converted to a series for subsequent computations.

The mean and standard deviation of the series allow you to form a normal distribution, from which you can compute the Z-score of the observed Sub-Saharan African sample.

```
s_size = subsaharan_data.shape[0] # Sample size
population = happy_full121["Life Ladder"]

# Sample from the dataset
sample_means = []
for _ in range(1000):
    sample = population.sample(n=s_size, replace=True)
    sample_means.append(sample.mean())
sample_means = pd.Series(sample_means)

# Calculate Z-score for the observed data
observed = (subsaharan_data.mean() - sample_means.mean()) \
            / sample_means.std()
pvalue = sts.norm.cdf(true_zscore)
print(sample_means.mean(), sample_means.std(), observed, pvalue)
#> 5.628 0.230 -4.970 7.679e-07
```

The Z-score’s low p-value obtained through the cumulative distribution function (CDF) confirms the rejection of the null hypothesis.

[Figure 4.6](#) on p. 73 illustrates the results by showing the close-to-normal distribution of the Z-score of the difference between the sample means and the observed mean.

The upcoming section provides another illustration of using loops in statistical research.

12.4 Automating Boring Tasks

Would you like to discover if there has been a statistically significant change in the Ladder score in any global region? This question can be answered by applying the `sts.ttest_rel()` function across all regions in the dataset. With the help of the Pandas function `Series.unique()` first introduced in Problem 13 on p. 233, you can remove duplicates from the series, obtain an alphabetical listing of all regions as `data["Region"].unique()`, and process them in a “for” loop. This quick procedure can instantly perform the t-test for each region, saving you considerable time. Here is how to do it:

```
for region in sorted(happy_12["Region"].unique()):
    d11 = happy_12.loc[(happy_12["Region"]==region) & \
                      (happy_12["year"]==2011), "Life Ladder"]
    d21 = happy_12.loc[(happy_12["Region"]==region) & \
                      (happy_12["year"]==2021), "Life Ladder"]
    result = sts.ttest_rel(d11, d21)
    print(f'{region:30}{result.pvalue:.5f}')
```

The formatting specifications after the colons in the `print()` function control the appearance of the output. For instance, `{region:30}` ensures that the `region` value displays with at most 30 characters, left-justified, and if needed, padded on the right with spaces. `{result.pvalue:.5f}` prints the p-value with exactly five digits after the decimal point.

You might notice the shift from the dataframe `happy_full` to `happy_12`. This alteration is necessary because the `sts.ttest_rel()` function requires pairwise-organized samples with corresponding values in both sets. However, the World Happiness Report cites 146 countries in 2011 and only 119 in 2021. To apply `sts.ttest_rel()`, you must find the intersection of these country sets using the NumPy function `numpy.intersect1d(x,y)`. The resulting `happy_12` dataframe is thus a balanced subset concerning 2011 and 2021.

```
c11 = happy_full.loc[happy_full["year"]==2011, "Country name"]
c21 = happy_full.loc[happy_full["year"]==2021, "Country name"]
c = np.intersect1d(c11, c21)
print(len(c))
#> 112
happy_12 = happy_full[happy_full["Country name"].isin(c)]
```

Remember that the balanced dataframe contains only 112 out of 166 countries mentioned in the full report. The experiment’s findings are insightful but only partially accurate.

#> East Asia & Pacific	0.79472
#> Europe & Central Asia	0.00002
#> Latin America & Caribbean	0.36331

#> Middle East & North Africa	0.17928
#> North America	0.30244
#> South Asia	0.27121
#> Sub-Saharan Africa	0.84707

A p-value of 0.00002 shows that the Ladder score in the “Europe & Central Asia” region may have changed, but it remained statistically unchanged in all other areas.

12.5 What’s Next

In the upcoming chapter, you will craft your custom functions, enabling you to develop a personalized toolkit. Utilizing functions in Python, instead of mere code fragments, offers the benefits of modularity, reusability, and better organization. This structured approach simplifies complex tasks and promotes efficient coding practices. Additionally, you will master linear regressions, enhancing your data modeling and prediction skills.

12.6 Programming Exercises

Exercise 14. Rewrite the bootstrapping code on p.246 using a “while” loop instead of the “for” loop.

Exercise 15. According to the Shapiro–Wilk test, which numerical variables in the World Happiness Report have a close-to-normal distribution in 2021? You can use the function `DataFrame.select_dtypes(include=np.number)` to extract only the numeric columns.

Exercise 16. Use a t-test to validate the hypothesis that the “Perceptions of corruption” and “Freedom to make life choices” variables in the World Happiness Report were independent in 2021. Apply the test only to the countries for which both variables are defined.

Exercise 17. Use a one-way ANOVA to investigate if the number of countries and territories included in the World Happiness Report yearly depends on the region.

User-Defined Functions and Regression Analysis

Before proceeding to regression analysis, you must learn another crucial Python concept: user-defined functions.

13.1 Writing User-Defined Functions

As introduced in [Chapter 9](#), a function is a cohesive, named unit of computation. Each function possesses a name, a clear purpose, and a way to be used (called). You are already acquainted with functions like `float()`, `int()`, `print()`, `len()`, and `str()`, among others. Functions offer structured and layered programming by presenting higher-level constructs and concealing the intricacies of their workings. While crafting a complex program without employing functions is technically feasible, doing so would render the codebase nearly incomprehensible and challenging to adapt or enhance.

Take, for instance, a hypothetical function `linear_regression()` designed for linear regression tasks. Crafting this function in pure Python is not trivial; it demands profound domain expertise and proficiency in the language. You could create a slower and less precise initial version and refine it subsequently or await a superior version from another developer. Once a better implementation emerges, the old function can seamlessly be supplanted by the new one without necessitating changes in the rest of the program.

Functions with intuitive names, such as `linear_regression()`, naturally serve as self-comments, reminding the reader of what actions are executed and in what order.

Internally, a function is a sequence of statements that can accept arguments, conduct specific tasks, and yield a result. Once defined, this sequence can be executed multiple times throughout different sections of your code. The process of crafting a new function entails the following:

1. Selecting an appropriate name for the function,
2. Defining its interface, which encompasses the arguments and the return value,

3. Drafting the body of the function, and
4. Documenting the design and intent.

Let's revisit the code fragment on p. 239 and convert it into a function that calculates the mean length of all words in a sentence.

```
words = sent.split()
lengths = [len(w) for w in words] # No condition!
mean_length = sum(lengths) / len(lengths)
print(mean_length)
```

13.1.1 Choose the Function Name

A function name should adhere to the same conventions as a variable name (p. 172). It should:

- Start with a letter or an underscore.
- Consist of letters, digits, and underscores.
- Be easily typable.
- Explain the function's purpose.

Though not mandatory, it is a common practice to incorporate verbs in function names, capturing their active role. For instance, a function designed to fetch data might be aptly named ``read_data``, and a function performing linear regression would be named ``run_linear_regression``, though ``linear_regression`` alone is usually sufficient. In this spirit, the new function can be called ``get_mean_length``, ``mean_length``, ``get_mean_word_length``, or ``mean_word_length``. The last variant is the cleanest and reasonably compact.

A function definition starts with the keyword ``def``. The keyword is followed by the function name, the list of arguments (explained later), and a semicolon. The remaining lines constitute the function body. All statements of the body must be indented by the same amount:

```
def mean_word_length(...something...):
    # Function body here

# Call the function here or elsewhere
```

The blank line at the end of a definition is not required but makes the program text more readable.

13.1.2 Define the Interface

The interface of a function encompasses its arguments (if any) and the result it returns. Certain functions necessitate additional information to execute the desired task. Such information is termed arguments. For instance, the `print()` function displays a message but requires clarity on what message to display. That specific message acts as an argument. When the `print()` function is invoked without arguments, it outputs the most uncontroversial message, an empty string.

Often, you can directly proceed to define a function without specifying any arguments. Python will alert you with a `NameError` if the function requires further data. In the context provided, your newly defined function requires the `sentence` argument to specify the sentence it should split.

```
def mean_word_length(sentence):  
    # Function body here
```

The argument `sentence` is placed within parentheses. In this context, it is treated as a generic placeholder (“a sentence”) rather than a specific entity (“the sentence” inside the function body). The exact value of the argument is not determined at the time of function definition. It might be “Mary Had a Little Lamb,” an empty string, or even the entirety of Shakespeare’s works. The actual value is identified when the function is called, and the function should be designed to handle any string input¹.

For functions necessitating multiple arguments, the arguments are separated by commas:

```
def add_a_and_b(a, b):  
    # Function body here
```

Even if a function does not need any arguments, its definition mandates the inclusion of empty parentheses:

```
def print_help_message():  
    # Function body here
```

As previously mentioned, if you attempt to use an argument without a prior definition, Python will generate an error.

A function processes input to produce a result. This result can be displayed, returned to the calling function (like the command line), or both. While the `print()` function lets you display the result, it does not allow for its subsequent reuse. Specifically, you cannot assign the displayed result to a variable for further calculations. Use the `return` statement in your function to enable such reuse.

¹Polymorphic functions can accept arguments of varied types. They must verify the type of the input argument before processing.

The ``return`` statement always terminates the execution of the function and has three primary forms:

- ``return x``, where ``x`` is a valid Python expression, sends the value of ``x`` back to the calling function. You can imagine this as replacing the function call with the value of ``x``.
- Just ``return`` without an accompanying expression will return ``None``.
- If a function completes without encountering a ``return`` statement, it will also return ``None`` by default.

Consider the code segment calculating the mean word length. If you wish to use this computed value in linear regression analysis, merely displaying it would necessitate manual re-entry. A more efficient approach is to return the value:

```
def mean_word_length(sentence):  
    # Function body here  
    return mean_length
```

With this setup, you can capture, store, and use the returned mean length in subsequent computations.

13.1.3 Implement the Function Body

The core of any function lies in its body, which comprises a series of statements aligned to an algorithm (the problem-solving recipe). Often, transforming existing code into a function is as straightforward as indenting, renaming variables as required, and adding the ``def`` keyword at the beginning, followed by a ``return`` statement at the end.

```
def mean_word_length(sentence):  
    words = sentence.split()  
    lengths = [len(w) for w in words]  
    mean_length = sum(lengths) / len(lengths)  
    # Uncomment below to print the result  
    # print(mean_length)  
    return mean_length
```

Here are key points to understand about the newly defined function:

- The term ``sentence`` in the function header represents any text passed in by the caller. Within the function, ``sentence`` refers to this same text.
- A user-defined function can invoke other functions, whether part of Python's standard library or user-defined.

- By default, the function will not display its output. However, you can easily change that by uncommenting the ``print()`` line.
- An edge case to be aware of is that if the input sentence consists solely of spaces, the function will throw a ``ZeroDivisionError``. This is an expected outcome, given the current design.

To verify that your function is defined correctly, type its name in the command line. Python will display the function's name and a memory address, which is not generally useful for developers.

```
mean_word_length
#> <function mean_word_length at 0x7f42e74a0378>
```

13.1.4 Invoking a User-Defined Function

Invoking a user-defined function mirrors calling any built-in or library function:

```
text = 'Linear regression is useful'
result = mean_word_length(text)
print(result)
#> 6.0
```

When calling a function, ensure the number and type of arguments you pass align with the function's expected arguments. It is worth noting that the argument names in the function call do not necessarily have to match those in the function definition; arguments are primarily matched by their position, not their name².

Perhaps unexpectedly, an attempt to access the variable ``words`` defined within the function results in an error:

```
words
#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> NameError: name 'words' is not defined
```

This error arises because all variables and arguments within a function are treated as local by default. They come into existence when the function is invoked and are discarded after it concludes. This encapsulation ensures that two distinct functions can use variables of identical names without conflict. For example:

```
def foo():
    var = 1
```

²Though Python supports passing arguments by name, that topic is beyond the scope of this book.

```

    bar()
    print(f'In foo: {var}')

def bar():
    var = 2
    print(f'In bar: {var}')

foo()
#> In bar: 2
#> In foo: 1

```

Both `foo()` and `bar()` contain a local variable named `var`. However, the modification of `var` in `bar()` does not impact the value of `var` in `foo()`.

Conversely, global variables are accessible and mutable both within and outside functions. They can be explicitly declared using the `global` keyword:

```

def foo():
    global var
    var = 1
    bar()
    print(f'In foo: {var}')

def bar():
    global var
    var = 2
    print(f'In bar: {var}')

foo()
#> In bar: 2
#> In foo: 2

```

Though global variables offer broad access, over-reliance on them can introduce ambiguous dependencies, complicating debugging and modification efforts. General software development discourages excessive global variables, but they can sometimes be suitable for specific tasks, like data science and statistical projects.

The next code fragment illustrates one more argument passing nuance. The function `reverse_string()` reverses the string `s` using the slicing operator and changes the value of the argument `status` to `True` to report success:

```

def reverse_string(s, status):
    s = s[::-1]
    status = True

text = 'Linear regression is useful'
reverted = False
reverse_string(text, reverted)
print(text, reverted)

```

#> Linear regression is useful False

The function did not perform as intended. In Python, immutable arguments: booleans, numbers, strings, and tuples—are passed by value. It means the function receives a copy of the original argument. Inside ``reverse_string()``, ``s`` is a copy of ``text``, and ``status`` is a copy of ``reverted``. Any modifications made inside the function do not impact the original variables. To alter the original variable, you must either declare it as global or return the modified value and reassign it outside the function.

Contrastingly, mutable arguments, such as lists, sets, and dictionaries, are passed by reference. Changes made to these arguments within a function directly affect the original variables. The ``add1()`` function below updates the dictionary ``d``, and as a result, it simultaneously modifies the ``people`` dictionary outside the function because both refer to the same dictionary.

```
def add1(d, key):
    if key in d:
        d[key] = d[key] + 1
    else:
        d[key] = 1

people = {} # Create an empty dictionary
add1(people, 'man')
add1(people, 'man')
add1(people, 'woman')
print(people)
#> {'man': 2, 'woman': 1}
```

Remember to maintain consistent indentation within the function to ensure readability and avoid syntax errors. Indentation should be relative to the function definition, not the page or surrounding context.

13.1.5 Document the Function Design

Your function definition is nearly complete. That is all there is to straightforward functions like ``mean_word_length``. However, for more intricate functions, consider adding a docstring. A docstring briefly describes the function's intent, parameters, and return values and occasionally offers usage examples. Place the docstring immediately after the function declaration, preceding the function's logic. Use triple quotes for the docstring to accommodate line breaks.

```
def mean_word_length(sentence):
    """
    Calculate the average word length in a given sentence.

    Arguments:
    - sentence (str): The input string.
```

Returns:

– *float: The mean word length of the sentence.*

Raises:

– *ZeroDivisionError: If sentence consists only of spaces.*

Example:

```
>>> mean_word_length("Hello World")
5.5
"""
words = sentence.split()
lengths = [len(w) for w in words]
mean_length = sum(lengths) / len(lengths)
return mean_length
```

Once added, you can leverage Python's built-in help system via the ``help()`` function.

```
help(mean_word_length)
#> Help on function mean_word_length in module __main__:
#>
#> mean_word_length(sentence)
#>     Calculate the average word length in a given sentence.
#>
#>     Arguments:
#>     – sentence (str): The input string.
#>
#>     Returns:
#>     – float: The mean word length of the sentence.
#>
#>     Raises:
#>     – ZeroDivisionError: If sentence consists only of spaces.
#>
#>     Example:
#>     >>> mean_word_length("Hello World")
#>     5.5
```

Your future users, including yourself, will appreciate the hints!

13.2 Linear Regression

As discussed in [Chapter 5](#), linear regression is a technique used to model and analyze the relationships between a dependent variable and one or more independent variables. The goal is to find the linear relationship that best explains the data.

Python supports linear regression through various modules, each offering different functionalities:

- The ``numpy`` module provides the ``numpy.polyfit()`` function, which performs polynomial curve fitting using the Ordinary Least Squares (OLS) method to find the polynomial coefficients that best fit the data. The ``np.poly1d()`` function can convert these coefficients into a polynomial function. ``np.polyfit()`` does not provide any metrics to evaluate the quality of the fit, nor does it compute any statistics related to the regression.
- The ``scipy.stats`` module includes the ``sts.linregress()`` function, which calculates a first-order (linear) OLS regression. This function returns the slope and intercept of the regression line, and the R^2 value, p-value, and standard deviations for both the slope and intercept. However, this function does not support multivariate regression (refer to [Section 5.2](#) on p. 87 for more on this topic). Multivariate regression is provided by functions ``scipy.linalg.lstsq()`` or ``numpy.linalg.lstsq()``.
- The ``sklearn.linear_model`` module, part of the SciKit-Learn library, offers a comprehensive set of tools for linear regression, including the ``LinearRegression`` class. This class follows the SciKit-Learn convention of using ``fit()`` to compute the regression and ``predict()`` to make predictions based on the model. After fitting the model, you can access the regression coefficients through the ``coef_`` attribute and the R^2 value through the ``score()`` method. Unlike the previous modules, SciKit-Learn's ``LinearRegression`` can handle multivariate regression.
- Finally, the ``statsmodels.formula.api`` module provides a way to perform linear regression using an approach similar to the R programming language, complete with R-style formulas. It can be particularly appealing for users already familiar with R. This module is focused on statistical analysis but can also be used to make predictions.

Let us explore ``statsmodels.formula.api`` more closely.

13.2.1 Building Linear Regression Using Patsy

To construct linear regression models in Python, you can utilize the ``statsmodels.formula.api`` module, commonly imported as ``smf``. This allows for incremental model building by adding an intercept (the constant) and independent variables. A unique feature is its ability to use Patsy formulas, inspired by the R programming language, to define the regression.

Patsy: R-Style Formulas

Consider the World Happiness Report dataset for illustration. We will examine the relationship between the logarithm of GDP per capita and the ladder

score for 2021. First, load the dataset and filter data from 2021 (see Listing on p. 204). Next, build a formula.

In Python, a Patsy formula is a string, but it follows specific formatting rules:

- Variables in the formula should correspond to valid column names in a Pandas dataframe and be legitimate Python identifiers (variable names). If column names have special characters, use the Patsy quoting function `Q()`:

```
`Q('Life Ladder')`.
```

Quoting is unnecessary if a column name is a valid Python variable name.

- A formula starts with the dependent variable's column name, followed by a tilde:

```
`Q('Life Ladder') ~ ...`
```

- Subsequent terms represent independent variables, separated by plus signs, signifying a combination, not an arithmetic addition. For instance:

```
`Q('Life Ladder') ~ Q('Log GDP per capita')`
```

```
`Q('Life Ladder') ~ Q('Log GDP per capita') + year`
```

```
`Q('Life Ladder') ~ Q('Log GDP per capita') + year + ...`
```

- To exclude the intercept, subtract 1 from the formula, e.g.:

```
`Q('Life Ladder') ~ Q('Log GDP per capita') + year - 1`
```

- Directly apply NumPy functions to columns to prevent creating temporary ones, e.g.:

```
`Q('Life Ladder') ~ np.log(Q('Log GDP per capita'))`
```

- A notation exists for categorical variables (explained later).

Finally, convert the formula string into an Ordinary Least Squares (OLS) object using `api.ols()`:

```
import statsmodels.formula.api as smf
formula = "Q('Life Ladder') ~ Q('Log GDP per capita')"
data = happy21.dropna(subset=["Life Ladder",
                             "Log GDP per capita"])
model = smf.ols(formula, data=data)
```

Upon defining the model, it is ready for data fitting.

Model Fitting and Analysis

The `fit()` function facilitates data fitting. After fitting, retrieve the comprehensive regression summary using the `summary2()` function:

```
fitted_model = model.fit()
summary = fitted_model.summary2()
print(summary)
#> ...
```

This detailed summary appears across three subtables in [Table 13.1](#). It contains statistics galore, but the most important (highlighted in the table) are the sample size (“No. Observations”), the R-squared value, the intercept, and p-values (“P> |t|”), standard errors (“Std.Err.”), and coefficients (“Coef.”) for each independent variable.

You can export the summary in various formats, like CSV, HTML, and $\LaTeX$, for later reference. All functions in the following listing return appropriately formatted strings.

```
summary.as_csv()
summary.as_html()
summary.as_latex() # Used for the Table 13.1
```

TABLE 13.1
Results: Ordinary Least Squares

Model:	OLS	Adj. R-squared:	0.625
Dependent Variable:	Q('Life Ladder')	AIC:	234.1123
Date:	2023-10-28 14:46	BIC:	239.5313
No. Observations:	111	Log-Likelihood:	-115.06
Df Model:	1	F-statistic:	184.3
Df Residuals:	109	Prob (F-statistic):	3.53e-25
R-squared:	0.628	Scale:	0.47397

	Coef.	Std.Err.	t	P> t	[0.025	0.975]
Intercept	-2.5505	0.6064	-4.2057	0.0001	-3.7524	-1.3485
Q('log GDP...	0.8572	0.0631	13.5775	0.0000	0.7320	0.9823

Omnibus:	35.828	Durbin-Watson:	1.958
Prob(Omnibus):	0.000	Jarque-Bera (JB):	96.321
Skew:	-1.167	Prob(JB):	0.000
Kurtosis:	6.922	Condition No.:	90

13.2.2 Using the Fitted Model for Predictions and Visualization

A fitted model is versatile—it can explain existing data and predict new outcomes. With the ``summary`` from the fitted model, you can utilize the ``predict()`` function in two main ways:

In-sample Predictions: When the ``predict()`` function is invoked without arguments, it returns predictions for the data points initially used to train the model. The returned predictions are structured as a NumPy array.

```
predicted = fitted_model.predict()
```

Out-of-sample Predictions: If you provide a dataframe (either the original or a new one with matching column names) as an argument to the ``predict()`` function, it will produce predictions for that specific dataframe. These predictions are returned as a Pandas series.

```
predicted = fitted_model.predict(data)
```

Residuals, which are the differences between the true values of the dependent variable and the model's predictions, offer insight into the model's accuracy. Visualizing these residuals can be exceptionally informative.

For instance, plotting residuals against an independent variable can reveal patterns in the data. A sample visualization might resemble [Figure 5.6](#) on p. 84.

```
residuals = data["Life Ladder"] - predicted
plt.scatter(x=data["Log GDP per capita"], y=residuals)
```

In the above code, residuals are computed by subtracting the predicted values from the actual “Life Ladder” values, and then they are plotted against “Log GDP per capita.”

13.2.3 Conducting Linear Multiple Regression using Patsy

A multiple regression involves more than one independent variable. While the procedure for fitting, analyzing, and visualizing the data remains unchanged, the formula for defining such a regression with Patsy requires additional components.

Let's investigate the impact of variables “Log GDP per capita,” “Freedom to make life choices,” “Generosity,” “Perceptions of corruption,” and “Confidence in national government” on the “Life Ladder.” While manually writing the Patsy formula is possible, dynamic situations may demand an automated approach, especially when dealing with multivariable regression. Fortunately, a formula is a string you can construct piece by piece, as demonstrated below:

```
# Specify independent and dependent variables
indeps = ["Log GDP per capita", "Freedom to make life choices",
          "Generosity", "Perceptions of corruption",
          "Confidence in national government"]
dep = "Life Ladder"

# Handle missing data
all_vars = indeps + [dep]
data = happy21[all_vars].dropna()

# Construct the formula
formula = f"Q('{dep}')~"
formula += "+".join(f"Q('{c}')" for c in indeps)
print(formula)
#> Q('Life Ladder')~Q('Log GDP...')+...+Q('Confidence in... ')

# Initialize an OLS model
model = smf.ols(formula, data=data)

# Fit the model and obtain a summary
fitted_model = model.fit()
summary = fitted_model.summary2()
# summary.tables[1] is a DataFrame
print(summary.tables[1][["Coef.", "P>|t|"]])
#>
#>      Coef.      P>|t|
#> Intercept      -2.919299  1.078704e-02
#> Q('Log GDP per capita')  0.639771  1.386826e-13
#> Q('Freedom to make life choices')  4.391540  5.017885e-10
#> Q('Generosity')  0.493438  2.020886e-01
#> Q('Perceptions of corruption') -0.795433  9.080088e-02
#> Q('Confidence in national government') -0.928908  3.858523e-02
```

Note that the summary consists of three tables stored as a list of Pandas dataframes. You can individually access them to extract relevant insights. For instance, here is how to identify statistically significant independent variables:

```
MAX_PV = 0.05
sig = summary.tables[1].query(f"P>|t|`<{MAX_PV}").index.tolist()
print(sig)
#> ['Intercept', "Q('Log GDP per capita')",
#>  "Q('Freedom to make life choices')",
#>  "Q('Confidence in national government')"]
```

Use backticks with `P>|t|` since it is not a valid Python variable name.

For a comprehensive interpretation of these results, refer to [Table 5.4](#) on p. 89 and the associated text.

13.2.4 Incorporating Categorical Variables

In our earlier discussions, we primarily focused on continuous variables. However, many datasets also include categorical variables.

For linear regression, a categorical variable with N levels translates to $N-1$ binary “dummy” variables. Each dummy variable corresponds to one level of the categorical variable. If a particular level is active, its corresponding dummy is 1; otherwise, it is 0. If all dummies are 0, then the categorical variable is assumed to be at its base level.

You can easily convert a categorical variable into dummy variables using the ``pandas.get_dummies()`` function. This function returns a dataframe of dummy variables. If you wish to exclude the first level (to serve as the base level), you can set the ``drop_first`` argument to `True`. To differentiate dummy columns, you can provide a prefix and a separator using the ``prefix`` and ``prefix_sep`` arguments.

```
regions = pd.get_dummies(happy21["Region"], drop_first=True,
                        prefix_sep=".", prefix="Region")
print(regions.shape)
#> (119, 6)
# 119 rows, 6 columns for 7 levels
```

To prepare your data for regression, join the dummy variables with your original dataframe:

```
data = happy21.join(regions)
print(data.columns.values)
#> ['Country name' 'year' 'Life Ladder' 'Log GDP per capita'
#> 'Social support' 'Healthy life expectancy at birth'
#> 'Freedom to make life choices' 'Generosity'
#> 'Perceptions of corruption' 'Positive affect'
#> 'Negative affect' 'Confidence in national government'
#> 'Region' 'IncomeGroup' 'SpecialNotes'
#> 'Region.Europe & Central Asia'
#> 'Region.Latin America & Caribbean'
#> 'Region.Middle East & North Africa' 'Region.North America'
#> 'Region.South Asia' 'Region.Sub-Saharan Africa']
print(data.shape)
#> (119, 21)
# Now, 21 columns
```

In the above example, the “East Asia & Pacific” region serves as the baseline and, thus, does not have a corresponding dummy column.

An alternative way to handle categorical variables is by using Patsy’s ``C()`` function directly within the formula. This method is more streamlined as Patsy automatically manages dummy variables for you, including the base-level selection.

```
cont_indeps = ["Log GDP per capita", "Generosity",
               "Freedom to make life choices", "Perceptions of corruption",
               "Confidence in national government"]
categ_indeps = ["Region",]
dep = "Life Ladder"
formula = f"Q('{dep}')~"
formula += "+".join(f"Q('{c}')" for c in cont_indeps) + "+"
formula += "+".join(f"C(Q('{c}'))" for c in categ_indeps)
model = smf.ols(formula, data=data)
```

With this approach, Patsy elegantly handles the conversion and inclusion of categorical variables in your regression model.

13.2.5 Incorporating Interaction Terms

An interaction refers to a situation where the effect of one independent variable on the dependent variable differs depending on the level of another independent variable. Essentially, the relationship between an independent variable and the dependent variable is influenced by other independent variables.

Within the Patsy framework, interaction terms can be represented in two distinct manners: as a “product” or a “ratio.” The “ratio” format ``A:B`` includes an interaction term for the variables A and B in the equation. Conversely, the “product” format ``A*B`` incorporates the interaction term alongside individual terms for A and B. It is analogous to writing ``A+B+A:B``.

The variables in these interactions can either be continuous or categorical. When using Patsy, the distinction between them hinges on the use of the function ``C()``. Let’s explore two illustrative examples:

- Formula:
``Q('Confidence ...') ~ Q('Social support') * Q('Perceptions ...')``

Result:

	Coef.	P> t
Intercept	2.4843	<u>0.0000</u>
Q('Social support')	-1.7952	<u>0.0026</u>
Q('Perceptions of corruption')	-2.1155	<u>0.0020</u>
Q('Social support'):Q('Perceptions of corruption')	1.7013	<u>0.0251</u>

- Formula:
``Q('Confidence ...') ~ C(Region) * Q('Perceptions ...')``

Result:

	Coef.	P> t
Intercept	0.7856	<u>0.0000</u>
C(Region)[T.Europe & Central Asia]	0.0353	0.7708
C(Region)[T.Latin America & Caribbean]	0.6549	<u>0.0404</u>
C(Region)[T.Middle East & North Africa]	0.9574	<u>0.0037</u>
C(Region)[T.North America]	0.0859	0.8084
C(Region)[T.South Asia]	1.8047	<u>0.0145</u>
C(Region)[T.Sub-Saharan Africa]	0.5643	<u>0.0286</u>
Q('Perceptions of corruption')	-0.4537	<u>0.0052</u>
C(Region)[T.Europe & ...]:Q('Perceptions ...')	-0.0684	0.7051
C(Region)[T.Latin ...]:Q('Perceptions ...')	-0.9015	<u>0.0300</u>
C(Region)[T.Middle ...]:Q('Perceptions ...')	-1.1268	<u>0.0083</u>
C(Region)[T.North America]:Q('Perceptions ...')	-0.2263	0.7192
C(Region)[T.South Asia]:Q('Perceptions ...')	-2.0143	<u>0.0234</u>
C(Region)[T.Sub-Saharan ...]:Q('Perceptions ...')	-0.5300	0.1159

The notation like ``C(Region)[T.South Asia]`` denotes the categorical variable “Region” at the level “South Asia.” Statistically significant ($(P > |t|) \leq 0.05$) variables and their interactions are underlined. You can obtain individual coefficients and p-values for each variable, level, and interaction from the dictionaries ``model.params`` and ``model.pvalues``:

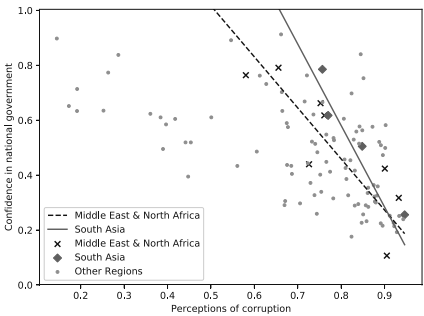
```
level = 'C(Region)[T.Sub-Saharan Africa]'  
print(model.params[level])  
#> 0.5643445075614467  
print(model.pvalues[level])  
#> 0.028633800192744165
```

For further insights on understanding the tables, refer to [Subsection 5.2.2](#) on p. 92.

13.3 Visualizing Interaction Effects

This section is dedicated to enhancing your understanding of regression models through effective visualization techniques. Here, you will learn to combine line and scatter plots with model coefficients to bring regression results to life.

For an illustrative example, refer to the Figure on p. 5.14 on p. 98, also presented to the right at half its original size for your convenience.



You have already met `matplotlib`, a comprehensive library for creating static, animated, and interactive visualizations in Python, in [Subsection 11.3.3](#) on p. 224. Assuming you have loaded the dataset, selected 2021 data, and addressed missing values (if not, refer to the instructions on p. 204), your data should now be in the `happy21` dataframe.

The upcoming code segment sets up the independent variable `ind` and two dependent variables: `dv1` (a categorical variable with two key levels `dv1_l1` and `dv1_l2`), and `dv2` (a continuous variable). Ensure these variable names match those in your dataset columns. Note that `dv1_l0` is used solely for the graph's legend. This setup allows for visualizing the interaction of any three variables at two levels.

Listing 13.1

interaction.py

```
# Set up the variables and levels
dv1    = 'Region'
dv1_l0 = 'Other Regions'
dv1_l1 = 'Middle East & North Africa'
dv1_l2 = 'South Asia'
dv2    = 'Perceptions of corruption'
ind    = 'Confidence in national government'
```

Next, construct a `patsy` formula, transform it into an Ordinary Least Squares (OLS) model, and fit this model. You can then extract crucial model components: slopes and intercept (for line plotting)—and segment the original dataset by levels for scatter plotting. The subset `subset0` contains rows not belonging to any specified level, identified using the `not in` operator in Python (see [Subsection 11.1.1](#) on p. 218).

Listing 13.2

interaction.py (continued)

```
# Prepare and fit the OLS model
formula = f'Q("{ind}") ~ Q("{dv2}") * C({dv1})'
model = smf.ols(formula, data=happy21).fit()

# Extract the slopes and the intercept
dv2_a    = model.params[f'Q("{dv2}")']
dv1_l1_a = model.params[f'C({dv1})[T.{dv1_l1}]']
dv1_l2_a = model.params[f'C({dv1})[T.{dv1_l2}]']
dv2_l1_a = model.params[f'Q("{dv2}") : C({dv1})[T.{dv1_l1}]']
dv2_l2_a = model.params[f'Q("{dv2}") : C({dv1})[T.{dv1_l2}]']
b        = model.params['Intercept']

# Split the data by region
subset1 = happy21.query(f'{dv1}=="{dv1_l1}"')
subset2 = happy21.query(f'{dv1}=="{dv1_l2}"')
subset0 = happy21.query(f'{dv1} not in ["{dv1_l1}", "{dv1_l2}"]')
```

The final part of the script is about visualization.

Choose a style, such as `'grayscale'`, for specific needs like black-and-white printing. Determining the data's vertical and horizontal range, while not mandatory, optimizes the chart's size.

Using `plt.scatter()`, create three overlaid scatter plots, each differentiated by marker styles: crosses, diamonds, and dots. The `plt.plot()` function is then used for line plotting through two points whose X coordinates are the range of the continuous dependent variable `dv2` ("Perception of Corruption"). The Y coordinates are computed from the regression coefficients and intercept.

Listing 13.3

interaction.py (continued)

```
# Prepare the stylesheet
plt.style.use('grayscale')

# Find the extents
plt.ylim(ymin=1.1 * happy21[ind].max())
xmin = happy21[dv2].min()
xmax = happy21[dv2].max()

# Plot the original data
plt.scatter(subset1[dv2], subset1[ind], marker='x', label=dv1_l1)
plt.scatter(subset2[dv2], subset2[ind], marker='D', label=dv1_l2)
plt.scatter(subset0[dv2], subset0[ind], marker='.', label=dv1_l0)

# Plot the regression results
plt.plot((xmin, xmax),
         (dv2_a * xmin + dv1_l1_a + b,
          dv2_a * xmax + dv1_l1_a + dv2_l1_a + b),
         linestyle = '--', label = dv1_l1)
plt.plot((xmin, xmax),
         (dv2_a * xmin + dv1_l2_a + b,
          dv2_a * xmax + dv1_l2_a + dv2_l2_a + b),
         label = dv1_l2)

# Decorate
plt.xlabel(dv2)
plt.ylabel(ind)
plt.legend()
```

Your figure is ready to be displayed and saved as an image file.

13.4 What's Next

OLS linear regressions form the foundation of predictive analytics, paving the way for more complex models that handle various data and prediction tasks.

This next chapter will introduce you to classification algorithms, beginning with logistic regression, a method that bridges the understanding of linear models with the challenges of categorical prediction.

13.5 Programming Exercises

Exercise 18. Create a function `continuous_vars(df)` that, given the dataframe `df`, returns a list of names of all continuous variables based on their data types. In this context, continuous variables are defined as columns with data types of floating point numbers (`"float"` or `"float64"`) or integers (`"int"` or `"int64"`). Hint: use the attribute `DataFrame.dtypes` or the function `DataFrame.select_dtypes()`.

Exercise 19. Create a function `significant_vars(df, dep, lvl)` that performs Ordinary Least Squares (OLS) linear regression to model the relationship between a specified dependent variable `dep` and all other variables in a given dataframe `df`. The function should return the names of the independent variables that are statistically significant based on a provided significance level `lvl`. The function should use the OLS implementation from `statsmodels.formula.api`. All column names in the dataframe are considered valid Python variable names. All variables are assumed to be continuous. The dependent variable is guaranteed to be present in the dataframe. The intercept's significance should not be considered; even if it is significant, it should not be reported. Example of a function call: `significant_vars(df, 'y', 0.05)`.

Exercise 20. Analyze the impact of generosity on the Life Ladder over time, using data from the World Happiness Report and an OLS linear regression model. Specifically, investigate how the relationship between generosity and the Life Ladder evolves. Hint: look at the interaction between generosity and year.

Classification is assigning observations to a set of categories (classes). Common classification algorithms include logistic regression, neural networks, k-nearest neighbors (KNN), decision trees, random forest, and Naïve Bayesian. This chapter focuses on logistic regression and k-nearest neighbors classification.

14.1 Generators

The content of this chapter depends on the concept of generator expressions or simply generators. Generators are “lazy functions” that yield (“generate”) their return values piecewise as the caller needs them (thus the name). They significantly reduce the data footprint in memory, sometimes at the expense of a slight execution slowdown.

Consider a situation when you must calculate the mean values of several variables stored in a massive CSV file—say 16 billion rows. The “textbook” solution is to read the file as a Pandas DataFrame or Python list of lists, then call the Pandas function `.sum()` or Python function `sum()`. Either of these functions has an internal loop (see [Section 12.1](#) on p. 234) that iterates through the list or DataFrame and adds the current row of values to the previously accumulated sum. Note that:

- This “invisible” loop is already the second loop in the program. You employed the first one to read the raw data from the file.
- Each data row extracted from the file is used only once and is not needed after the addition but remains idle in memory unless you have further plans to reuse it.
- Thus, you use two loops of enormous length to create an unnecessary copy of the vast dataset, which may not even fit in the RAM.

A natural solution to the problems of duplicated data and loops is to pull rows from the file one at a time, process them (add to the totals), and

immediately discard them. In such a scenario, the object created and returned by the ``open()`` function, ``_io.TextIOWrapper``, acts as a generator¹:

```
with open("data.csv") as infile: # <class ' _io.TextIOWrapper'>
    for line in infile:
        do_something(line)
```

At each iteration of the loop, ``infile`` delivers the next line, which is assigned to the variable ``line``, used in the body of the loop, replaced by the next line at the end of the iteration, and discarded, saving memory space.

14.1.1 Built-In Generator Functions

The standard Python library provides several built-in functions that return generators.

Function `enumerate()`

Function ``enumerate(s, start=0)`` returns an ``enumerate object`` (a generator). The object yields tuples ([Subsection 10.1.1](#) on p. 190) containing a count (from ``start``) and the corresponding value from the sequence ``s``.

```
students = ["Alice", "Bob", "Chuck"]
e_students = enumerate(students)
for idx, name in e_students:
    print(f"{idx}\t{name}")
#> 0    Alice
#> 1    Bob
#> 2    Chuck
e_students = enumerate(students, start=1)
for idx, name in e_students:
    print(f"{idx}\t{name}")
#> 1    Alice
#> 2    Bob
#> 3    Chuck
```

The second code snippet starts the count from 1.

What happens if you try to reuse the ``enumerate`` object and walk over it again? Not much—actually, nothing:

```
for idx, name in e_students:
    print(f"{idx}\t{name}") # No output!
```

An ``enumerate`` object can be used only once. A yielded tuple is removed from the object. Either create a new generator whenever you need one or

¹ ``_io.TextIOWrapper`` acts like a generator but it is not truly a generator. It is an iterator. However, it is already familiar to you and serves as a convenient example.

convert the yielded tuples to a list and iterate over the list. However, the latter solution eliminates the advantages of the lazy generation.

```
l_students = list(enumerate(students))
print(l_students)
#> [(0, 'Alice '), (1, 'Bob '), (2, 'Chuck ')]
```

Function zip()

The function `zip(s1,s2,...,sn)` returns a `<zip object>` (a generator, too). The object yields `n`-length tuples containing the `i`-th items of each sequence `s1`, `s2`, ..., `sn`, starting at the beginning. The generation ends when the shortest sequence ends.

Function `zip()` is frequently used to bind the items of parallel lists: two or more lists of the same length such that the `i`-th items in each list refer to the attributes of the same object. Converting such lists into a list of tuples prevents them from inconsistent modifications.

```
levels = ["Freshman", "Sophomore", "Senior"]
stud_data = zip(students, levels)
for name, level in stud_data:
    print(f"{name}\t{level}")
#> Alice      Freshman
#> Bob        Sophomore
#> Chuck      Senior
stud_data = zip(range(len(levels)), students, levels)
for idx, name, level in stud_data:
    print(f"{idx}\t{name}\t{level}")
#> 0    Alice  Freshman
#> 1    Bob    Sophomore
#> 2    Chuck  Senior
```

You can convert a `zip` object to a list of tuples. You can also convert a `zip` object to a dictionary if only two lists are involved and all items on the first list are distinct. The first list becomes the list of keys, and the second is the list of values.

```
stud_dict = dict(zip(students, levels))
print(stud_dict)
#> {'Alice ': 'Freshman ', 'Bob ': 'Sophomore ', 'Chuck ': 'Senior '}
```

Function map() and Generator Chaining

Function `map(f,s1,s2,...,sn)` applies the function `f(x1,x2,...,xn)` to the corresponding items in the sequences `s1`, `s2`, ..., `sn`. The `<map object>` (a generator again) yields the application results one at a time.

```
name_lengths = map(len, students)
for name_len in name_lengths:
    print(f"{name_len}")
#> 5
#> 3
#> 5
```

You may want to display not only the name lengths but also the names of student IDs. For that, you need to chain the generators: feed the names and the output from ``map()`` to the input of ``zip()``:

```
print(list(zip(map(len, students), students)))
#> [(5, 'Alice '), (3, 'Bob '), (5, 'Chuck ')]
```

Think of the yielded sequences as data streams and generators as producers, transformers, and consumers of the streamed items. As a result, you can build complex and efficient data processing pipelines.

Function `filter()`

The last library generator function to explore is ``filter(f,s)``. The function returns a generator yielding those items of ``s`` for which ``f(item)`` is true. The function classifies the input sequence into “compliant” and “non-compliant.” The following example eliminates non-alphabetic strings from a list:

```
list(filter(str.isalpha, ["Hello", ",", "world", "!"]))
#> ['Hello ', 'world ']
```

14.1.2 Generator Expressions

Another class of generators is generator expressions. They look like list comprehensions ([Subsection 12.1.3](#) on p. 237) but are enclosed in parentheses rather than square brackets. The value of a generator expression is not a list but a generator, and the goal of the expression is to yield the items one by one, bypassing the list construction stage. The following combination of the function ``max()`` and a generator expression finds the student with the longest name:

```
print(max( ( len(name), name) for name in students) ))
#> (5, 'Chuck ')
print(max( [ len(name), name) for name in students] ))
#> (5, 'Chuck ')
```

The second example illustrates using a list comprehension to accomplish the same goal. It produces the same answer and takes approximately 8% less time, as promised at the beginning of the Section. However, the generator expression does not need to concern itself with the size of the list ``students``.

14.2 Classification via Logistic Regression

Despite what its name suggests, logistic regression is predominantly employed for classification purposes rather than for regression in the traditional sense. It is especially effective in binary classification tasks, where distinguishing between two possible outcomes, such as “success” vs. “failure” or class 1 vs. class 0.

14.2.1 Binary Logistic Regression

Logistic regression uses the logistic (or sigmoid) function to estimate the probability that a given observation falls into a specific class. The function returns a value between 0 and 1, representing the probability that the input belongs to the class labeled as 1. Internally, the algorithm calculates the weighted sum of the input features, including a bias or intercept term, and then applies the logistic function to this sum to estimate the probability.

The essence of logistic regression lies in its ability to provide a probability score that can be interpreted directly or converted into a binary category by applying a threshold, typically set at 0.5. If the calculated probability is above this threshold, the outcome is predicted as class 1; otherwise, it is predicted as class 0.

Two Python modules provide binary logistic regression tools: `Scikit-learn` (`sklearn.linear_model``, function `LogisticRegression()`) and `statsmodels` (`statsmodels.formula.api``, function `.logit()`). We will focus on the latter solution, which is conceptually more statistical than the former.

Thanks to Patsy, fitting a logistic regression is practically indistinguishable from fitting an OLS regression (see [Subsection 13.2.1](#) on p. 257). All you need to do is replace a call to `smf.ols()` with a call to `smf.logit()`.

To understand logistic regression, let’s classify the countries from the World Happiness Report dataset by income groups based on the perceptions of corruption in 2021. The dataset assigns each country to one of the four groups: Low income, Lower Middle income, Upper Middle income, and High income. However, logistic regression can do only binary classification. It is your responsibility as a researcher to group the four levels into two—for example, by combining the first and the last two levels.

```
# Data preparation
file_path = "data/happiness-full.csv"
data = pd.read_csv(file_path, index_col=0)\
    .query("year == 2021")\
    .rename(columns={"Perceptions of corruption":
                    "corruption"})\
    .dropna(subset=["corruption", "IncomeGroup"])

data["Upper"] = data["IncomeGroup"]\
```

```
.isin(("Upper middle income", "High income"))\
.astype(int)
```

The column “Upper” now contains 0 for the Low- and Lower-Middle-income countries and 1 for all other countries. The classification step is now trivial:

```
formula = "Upper ~ corruption"
model = smf.logit(formula, data).fit()
```

The fitted model parameters are available via `model.params`; the corresponding odds ratio is their exponent `numpy.exp(model.params)`:

```
print(model.params)
# Intercept      2.425831
# corruption    -2.703710
#dtype: float64
print(np.exp(model.params))
#Intercept      11.311620
#corruption      0.066957
# dtype: float64
```

You are advised to revisit [Subsection 6.1.1](#) on p. 103 and look at [Figure 6.2](#) on p. 104 for a more profound results analysis.

14.2.2 Multinomial Logistic Regression

Multinomial logistic regression ([Section 6.2](#) on p. 107) evaluates the log odds of an observation belonging to each category with respect to a chosen reference category. It is an extension of binary logistic regression used when the dependent variable is categorical with more than two levels.

Multinomial logistic regression is particularly useful when the categories are nominal (i.e., they do not have a natural ordering). The reference category is typically chosen either because it is the most common category or for some theoretical reason. The choice of reference category can affect the interpretation of the model coefficients but not the model’s ability to predict or classify observations.

You can still use Patsy to fit the multinomial logistic regression model. The model itself is implemented as `smf.mnlogit()`. The following formula investigates the effect of Life Ladder and Log GDP per capita in a country on the country’s association with one of the global regions (namely, “Latin America & Caribbean,” “Middle East & North Africa,” or “Sub-Saharan Africa”). The region names are first converted to numerical region codes, as Patsy expects.

Note the use of the `enumerate()` generator that arranges category names into a list of (number, name) tuples for further conversion into a dictionary. You will need the dictionary later to translate the codes back to human-readable labels.

```
cats = pd.Categorical(data["Region"], ordered= False)
data["RegionCode"] = cats.codes
code_dict = dict(enumerate(cats.categories))
print(code_dict)
# {0: 'East Asia & Pacific ', 1: 'Europe & Central Asia ',
# 2: 'Latin America & Caribbean ', 3: 'Middle East & North Africa ',
# 4: 'North America ', 5: 'South Asia ', 6: 'Sub-Saharan Africa '}

formula = "RegionCode ~ ladder + gdp"
model = smf.mnlogit(formula, data).fit()
```

Printing out the summary of the results `model.summary2()` produces copious output. The summary includes the table with the regression statistics and subtables with the model's coefficients for each region code. Let us visualize some results in the spirit of replacing a thousand words with a picture. Ideally, three-dimensional visualization would be more appropriate for three variables. However, since we are confined to a two-dimensional page, let's produce three two-dimensional snapshots for three levels of the Ladder score: 25th, 50th, and 75th percentiles.

```
# An array of X (GDP) values for prediction
gdp = np.linspace(data.gdp.min(), data.gdp.max(), 100)
quantiles = [0.25, 0.5, 0.75] # quartiles and the median
ladders = data.ladder.quantile(quantiles)

region_names = [ # The regions of interest
    "Latin America & Caribbean",
    "Middle East & North Africa",
    "Sub-Saharan Africa"
]
```

Lastly, prepare lists of line styles for the regions (based on the Matplotlib `'lines'` module), presentable subplot titles, and drawing axes.

```
styles = list(mplll.lineStyles.keys())
titles = [f"Ladder Score = {(q*100):.0f}th Percentile"
          for q in quantiles]
fig, ax = plt.subplots(len(region_names), 1, figsize=(8, 8))
```

At this point, you have three parallel lists: `'ax'` (of axes), `'titles'` (of titles), and `'ladders'` (of life ladders), which should be combined to produce consistent plots. The `'zip()'` generator function fits such a job. It traverses the lists in parallel in a loop and generates three-element tuples consisting of an axis, a title, and a ladder score. The loop's body predicts the country's probability of being in each selected region (`'predicted'`), plots the probabilities using different line styles, and provides the matching title.

```
for axis, title, ladder in zip(ax, titles, ladders):
```

```

new_data = {"ladder": np.full(gdp.size, ladder), "gdp": gdp}
predicted = model.predict(new_data).set_index(gdp)\
                .rename(columns=code_dict)[region_names]

predicted.plot(style=styles, ax=axis)
axis.set_title(title)
axis.set_ylim(0, 1)
axis.set_xlabel("Log GDP per Capita")
axis.set_ylabel("Estimated Probability")
axis.legend().set_visible(False)

```

Figure 6.3 on p. 110 shows the output of the script. Section 6.2 on p. 107 explains it in detail.

14.2.3 Ordered Logistic Regression

Ordered logistic regression is an advanced version of multinomial regression that incorporates an additional condition: the dependent variable levels must follow a specific sequence. This model assumes that while these levels are sequenced, their labels do not convey any numerical value beyond their inherent order.

Consider analyzing how life expectancy, freedom, and generosity affect different income categories, such as Low Income, Lower Middle Income, Upper Middle Income, and High Income. You can use Patsy to create the formula to set up the regression analysis, just as you would with other regression models. However, merely defining the formula is not sufficient. To proceed, you must transform this formula into an operational model via the `OrderedModel.from_formula()` method. For the fitting process, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is recommended due to its superior performance in handling nonlinear optimization tasks compared to the default Newton–Raphson method, making it more effective for the complexities of ordered logistic regression.

```

from statsmodels.miscmodels.ordinal_model import OrderedModel
...
formula = "IncomeGroup ~ life + freedom + generosity"
model = OrderedModel.from_formula(formula, data, distr="logit")
modelResult = model.fit(method="bfgs")

```

Table 6.3 on p. 112 shows the optimization results. The cutpoints that separate the four income group categories are of particular interest. You can identify the corresponding parameters by a slash “/” in their names and transform them for better interpretability.

```

params = modelResult.params
cutpoints = params[params.index.str.contains("/")]
print(model.transform_threshold_params(cutpoints))

```



```
# [      -inf 33.1912199  37.99590053 40.38091317      inf]
```

[Section 6.3](#) on p. 111 further explains the output of the optimization.

Following the discussion of the parallel regression assumption in [Subsection 6.3.1](#) on p. 112, you can examine the assumption by performing an ordered logistic regression and comparing its log-likelihood (model fitness) with a multinomial logistic regression with the same variables.

```
data["IncomeGroupMN_code"] = pd.Categorical(data["IncomeGroup"],
                                             ordered=False).cat.codes
formulaMN = "IncomeGroupMN_code ~ life + freedom + generosity"
modelMN = smf.mnlogit(formulaMN, data).fit()

G = -2 * (modelResult.llf - modelResultMN.llf)
df = modelResultMN.df_model - modelResult.df_model
p = 1 - chi2.cdf(G, df)
print(p)
# 0.09964240997879703
```

The non-significant difference between the models (p-value = 0.1) suggests that the two models have similar goodness-of-fit, as illustrated by [Figure 6.4](#) on p. 114.

14.3 K-Nearest Neighbors Classification

The k-nearest neighbors algorithm represents another class of classification algorithms. It does not make any assumptions about the data distribution. Instead, it assigns a label to the new data based on its similarity to the preclassified data.

While often used for classification, KNN can also be applied to regression problems by taking the average (or another aggregate measure) of the K nearest neighbors' values rather than voting on the most frequent label. KNN is also characterized as lazy because it does not learn a discriminative function from the training data but memorizes the dataset instead. The computation is deferred until a new observation must be classified.

While [Section 6.4](#) on p. 113 meticulously explains the algorithm, in this section, we are more interested in the technicalities of its application. The ``sklearn.neighbors`` module, part of the previously mentioned SciKit-Learn library, offers a ``KNeighborsClassifier()`` that encapsulates the algorithm and provides several optional configuration arguments, some of which, with their default values, are explained below:

- ``n_neighbors=5``—the number of neighbors;

- `metric="minkowski"`—the metric to compute distances;
- `p=2`—power parameter for the Minkowski metric (1 for Manhattan distance L1, 2 for Euclidean distance L2).

The expression `knn = KNeighborsClassifier()`, thus, creates a new KNN classifier that uses five neighbors and calculates Euclidean distance. You can start using the classifier immediately by providing a two-dimensional array `X` of independent variables and a vector `y` representing the dependent variable (in this example, “Healthy life expectancy at birth,” “Freedom to make life choices,” “Generosity,” and “IncomeGroup” in 2021, as per the World Happiness Report).

```
X = data[["life", "freedom", "generosity"]]
y = data["IncomeGroup"]
knn = KNeighborsClassifier()
knn.fit(X, y)
print(knn.score(X, y)) # An equivalent of R-squared
#> 0.7
y_predict = knn.predict(X)
```

However, doing so will likely produce distorted, if not erroneous, results.

14.3.1 Normalization

The primary offender here is scaling. Suppose the absolute range of one independent variable, `x`, is much larger than the ranges of the other independent variables. In that case, the Euclidian distance between two observations will reflect the distance between them in terms of `x`, minimizing the impact of other variables:

```
X.max() - X.min()
#> life          23.049999
#> freedom       0.542397
#> generosity    0.831359
```

The k-nearest neighbors algorithm is acutely sensitive to scaling imbalance. You must rescale the independent variables with one of the scalers from the `sklearn.preprocessing` module, such as `StandardScaler()`. The `StandardScaler()` normalizes a variable by subtracting its mean and dividing by the standard deviation. As a result, the mean of all normalized variables is 0, and their standard deviation is 1.

```
X = StandardScaler().fit_transform(X)
```

As a rule, you should apply this scaler only to the approximately normally distributed variables. You can validate their normality visually or by

running the Shapiro–Wilk test (pp. 65 and 239). A `MaxAbsScaler()` or `MinMaxScaler()` is more appropriate for non-normal variables².

14.3.2 Cross-Validation

Another potential problem of a KNN classifier is the choice of `k`. It can be addressed via cross-validation: splitting the dataset into `cv` random equally sized subsets (folds) and using `cv-1` folds to train the model and the remaining fold to test it³. You can cross-validate any classification algorithm that depends on hyperparameters (external configuration variables, such as `cv`). You can further tune the hyperparameters: vary them in a specific range and optimize some model performance metrics, such as accuracy (p. 119).

Both tasks: cross-validation and tuning—can be automated. Create a generic `KNeighborsClassifier()` and a `GridSearchCV()` exhaustive hyperparameter tuner from the `sklearn.model_selection` module. The tuner requires the estimator (`knn`), a parameter grid (a dictionary of hyperparameters and their ranges), the number of folds `cv`, the performance metric (e.g., `"accuracy"`), and the number of CPUs to engage (`-1` indicates using all available CPUs).

```
knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, {"n_neighbors": range(2, 10)},
                           cv=5, scoring="accuracy", n_jobs=-1)
```

Next, randomly partition the observations into two subsets: training set (`X_train`, `y_train`, 75%) and testing set (`X_test`, `y_test`, 25%). The `random_state` argument initializes the random number generator for reproducibility and should be passed only while debugging the script.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.25,
                                                    random_state=23)
```

Put the testing set aside (you will use it for validation) and concentrate on the training set.

The tuner `grid_search` is a model, too, and it can be trained (“fitted”) by calling the standard method `.fit()`. The result of training is a set of hyperparameters’ values optimizing the estimator accuracy.

```
grid_search.fit(X_train, y_train)
best_k = grid_search.best_params_["n_neighbors"]
```

²Incidentally, none of the variables in the example passes the test. The p-values of “generosity,” “freedom,” and “life” are 0.0294 (borderline), 0.0001, and 0.0012, respectively. A normalization technique that does not assume normality might be more appropriate for them.

³Cross-validation with `cv = 2` is known as hold-out validation and does not require the folds to have equal size.

```
best_accuracy = grid_search.best_score_

print(f"Best k value: {best_k}")
#> Best k value: 8
print(f"Best cross-validation accuracy: {best_accuracy:.2f}")
#> Best cross-validation accuracy: 0.68
```

You can now use the best hyperparameter `k` to perform KNN on the training and testing sets, obtain both scores, predict the test outcomes `y\_predict`, and compare them to the true outcomes in `y\_test`.

```
knn.set_params(n_neighbors=best_k)
knn.fit(X_train, y_train)
print(knn.score(X_train, y_train))
#> 0.7804878048780488
print(knn.score(X_test, y_test))
#> 0.4642857142857143
y_predict = knn.predict(X_test)
```

The obtained scores are low, especially the testing score. The discrepancy between the scores may indicate overfitting: producing a model that corresponds too closely to the training data set but fails to fit additional data or predict future observations reliably. A confusion matrix (p. 117) and a classification report (a confusion matrix with row and column headers and calculated performance metrics) may shed some light on the source of the problem.

```
print(confusion_matrix(y_test, y_predict))
#> [[4 0 0 2]
#>  [0 1 0 0]
#>  [1 4 3 2]
#>  [4 1 1 5]]
print(classification_report(y_test, y_predict))
#>
#>
#> precision recall f1-score support
#>
#> High income 0.44 0.67 0.53 6
#> Low income 0.17 1.00 0.29 1
#> Lower middle income 0.75 0.30 0.43 10
#> Upper middle income 0.56 0.45 0.50 11
#>
#> accuracy 0.46 28
#> macro avg 0.48 0.61 0.44 28
#> weighted avg 0.59 0.46 0.47 28
```

The model strongly under-predicts Low-Income countries but fares well for both Middle-Income labels. The analysis of the last column in the table explains that the precision is correlated with the support. You may need to collect more data to improve both.

14.4 Receiver Operating Characteristic Curve

The function ``knn.predict()`` is designed to output binary classification results. However, rather than directly providing a binary outcome, it computes the likelihood of the outcome being positive (1) and then assigns a binary value based on a specified threshold. For a more transparent approach, the function ``knn.predict_proba(X)`` directly returns the probability of each outcome, allowing for more nuanced decision-making based on these probabilities.

Setting the threshold too high results in most outcomes being classified as negative (0), with a significant portion of these classifications being incorrect, leading to a high fall-out. On the other hand, setting the threshold too low causes most outcomes to be classified as positive (1), with many of these being incorrect, resulting in a high recall.

A Receiver Operating Characteristic curve (a ROC curve) is a visual tool used to evaluate the performance of a binary classifier system, such as the k-nearest neighbors classifier, across different threshold settings. It plots the recall (true positive rate) against the fall-out (false positive rate). Ideally, a ROC curve will rise sharply from the bottom left to the top left corner of the plot, then move directly to the top right corner, indicating a perfect classifier ([Figure 6.7](#) on p. 124). Any curve above the diagonal line indicates a model that performs better than random guessing, signifying a “good” model. Conversely, models that fall under this diagonal line are considered ineffective.

A model’s effectiveness is quantitatively assessed by the Area Under the Curve (AUC) metric associated with the Receiver Operating Characteristic (ROC) curve. An AUC value of 0 indicates an utterly ineffective model, 0.5 corresponds to a model whose performance is no better than random chance, and 1.0 signifies an ideal classifier that perfectly distinguishes between the classes.

You can utilize functions from the ``sklearn.metrics`` module to calculate the ROC curve and AUC. Below is an example of implementing this with a k-nearest neighbors (KNN) classifier in Python:

```
from sklearn.metrics import roc_curve, auc

classifier = KNeighborsClassifier()
classifier.fit(X_train, y_train)
y_probs = classifier.predict_proba(X_test)[: , 1]
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
roc_auc = auc(fpr, tpr)
```

The ``roc_curve()`` function computes the false positive rate (``fpr``) and true positive rate (``tpr``) at various threshold settings, along with the thresholds themselves. These can be plotted against each other to visualize the ROC curve. The ``auc()`` function calculates the AUC by integrating the area under the ROC curve.

Visualizing the ROC curve ([Figure 6.8](#) on p. 125) and understanding the AUC value are powerful methods for evaluating the performance of binary classification models. By adjusting the threshold, you can balance recall and fall-out to meet the specific needs of your application, with the ROC curve and AUC guiding the model's overall capability to distinguish between the two classes.

14.5 What's Next

This chapter concludes the “classical” statistical core of the book, equipping you with foundational statistical and machine-learning tools. The next chapter introduces network analysis, shifting your focus from individual data points to the intricate web of relationships in social networks, offering new insights into the dynamics of social structures and interactions.

14.6 Programming Exercises

Exercise 21. Write a complete Python program that prints every odd-numbered line in a large text file that does not start with a pound sign. The program must proceed line by line without loading the entire file into memory. The program must use generators.

Exercise 22. Repeat the KNN exercise (with cross-validation) by combining two lower- and two higher-income categories, as in [Subsection 14.2.1](#) on p. 272. Has the accuracy of classification improved?

More Generators and Network Analysis

In this chapter, you will learn how to automate complex network construction and discovery. While some complex networks are small (the “Karate Club” in [Figure 7.1](#) on p. 128), others are large (a planetary-scale social network of 240 million people in [\[LH14\]](#)). While some complex network operations are simple (getting the number of nodes), others are complex, for the lack of a better word (calculating eigenvector centralities, [Subsection 7.3.4](#) on p. 137). Applying a complex operation to a large network may result in an unsustainable amount of data, and the standard generators may be of little use due to the unusual nature of the problem. It is time to learn how to write your own generators.

15.1 Generators, Revisited

You have already seen the standard generators: `enumerate()`, `zip()`, `map()`, and more—in [Section 14.1](#) on p. 268. What you did not see is how to create a custom generator.

A generator is a “lazy function” that yields (“generates”) a fragment of the return value on demand. Each successive call to the generator yields the next fragment of the result, until the sequence of fragments ends. The interaction with the caller is handled by the keyword `yield`.

The difference between the statements `yield X` and `return X` is in the behavior of the function after the execution of the statements. There is no “after” for `return`: any statement following `return` is unreachable. Another call to a function with `return` starts at the beginning of the function, unaware of its execution history.

Conversely, the `yield` statement memorizes the line from which it was executed. When you call the enclosing generator function again, the execution continues at the next line. A generator may have several `yield` statements, each producing a result when executed. A generator may have a `yield` statement in a `for` loop, producing one result per iteration. Lastly, a generator may have a `yield` statement in a `while` loop, potentially yielding an infinitely long sequence of results.

Using a ``yield`` statement in a function does not preclude you from using a ``return`` statement. Just remember that ``return`` marks a definitive end of the function. There is no “after” after ``return``.

To illustrate hand-crafted generators, here is a function that generates consecutive integer numbers, starting from 0 and never stopping.

```
def integers():
    n = 0
    yield 0
    while True:
        n = n + 1
        yield n
    return # Unreachable

i_gen = integers() # A new generator
print(next(i_gen))
#> 0
print(next(i_gen))
#> 1
print(next(i_gen))
#> 2
```

Do not attempt to convert ``i_gen`` to a list of numbers! The result of the ``list(i_gen)`` is an infinite list, which needs infinite memory and takes infinite time to construct. As a side note, you can now build an infinite ``for`` loop:

```
for i in integers(): # This loop never ends
    do_something(i)
```

As you will see later in this chapter, generators are used in network analysis for graph traversals, finding paths, calculating centralities, subgraph extraction, and other tasks.

15.2 NetworkX and Graph Creation

A complex network may have a complex mathematical foundation. However, visually, it is most likely an elegant, often colored graph, and you can delegate its analysis to Python software, such as NetworkX [HSS08]¹, used in this chapter.

Mathematically, a network is a graph with attributes. NetworkX, usually imported as ``nx``, provides data structures for representing simple graphs

¹<https://networkx.org/>

(`Graph`, networks with only undirected edges and no parallel edges), digraphs (`DiGraph`, networks with directed edges and no parallel edges), multigraphs (`MultiGraph`, networks with parallel edges), and multidigraphs (`MultiDiGraph`). In addition to the data structures, NetworkX features a variety of graph algorithms, measurement and visualization tools, and generators for classic, random, and synthetic graphs.

Flexibility is an additional feature of NetworkX. It allows graph nodes to be purely mathematical (graph theoretical) objects, texts, images, or anything else that Python can represent. Likewise, edges can hold arbitrary attribute data not limited to floating-point weights.

15.2.1 Creating a Graph

NetworkX provides a variety of tools for graph construction.

Graph Constructors

Functions `nx.DiGraph()`, `nx.MultiGraph()`, `nx.MultiDiGraph()`, and `nx.Graph()` create the namesake graphs.

When you call a constructor without parameters, it returns an empty graph `G`. You can later expand it by adding individual nodes and edges (functions `Graph.add_node(n)`, `Graph.add_edge(n1, n2)`), lists of nodes and edges (functions `Graph.add_nodes_from(list_of_nodes)`, `Graph.add_edges_from(list_of_edges)`, or nodes or edges from another graph (`Graph.add_nodes_from(F)` and `Graph.add_edges_from(F.edges)`).

Alternatively, you can call the same function with an edge list or a node dictionary as a parameter. The new graph will consist of all the listed edges and the incident nodes in the former case or of the key nodes in the former case. If a graph has isolates, you should add them later.

The functions `Graph.remove_nodes()`, `Graph.remove_edges()`, `Graph.remove_nodes_from()`, and `Graph.remove_edges_from()` remove existing nodes and edges. When you remove an edge, its end nodes are not removed.

NetworkX represents an edge as a tuple of the start and end nodes and may optionally include the third element: a dictionary of edge attributes and their values.

The example below shows how to construct [a fragment of] the Zachary's Karate Club network using different node- and edge-handling techniques:

```
import networkx as nx

edges = [(0, 1, {"weight": 4}), (0, 2, {"weight": 5}),
         (0, 3, {"weight": 3}), (0, 4, {"weight": 3}), ...,
         (31, 33, {"weight": 4}), (32, 33, {"weight": 5})]
karate = nx.Graph(edges)
karate.add_node(17, club="Mr. Hi") # "club" is an attribute
karate.add_nodes_from([18, 19, 20])
```

```
karate.add_edge(0, 32, weight=3) # "weight" is an attribute
karate.add_edges_from([(4, 6, {"weight": 2}),
                       (4, 10, {"weight": 3})])
karate.add_node("Alice") # Oops...
karate.remove_node("Alice")
```

Graph Generators

The NetworkX submodule ``nx.generators`` features numerous network-generating functions. These are not genuine “lazy” generators but rather parametrized constructors of graphs of well-known types: balanced trees, complete graphs, cycle graphs, star graphs, etc.

From the Social Science perspective, some of the most useful graphs are:

- The Erdős–Rényi graph [ER59] explores network formation by randomly connecting nodes. Create it by calling ``nx.erdos_renyi_graph(nodes, p_edge)``, where ``nodes`` is the graph size and ``p_edge`` is the probability of connection between any two nodes.
- The Watts–Strogatz graph [WS98] introduces the concept of small-world networks with short average path lengths and high clustering coefficients, mimicking the “six degrees of separation” phenomenon often observed in social networks. Create it by calling ``nx.watts_strogatz_graph(nodes, edges_node, p_rewiring)``, where ``nodes`` is the graph size, ``edges_node`` is the initial number of edges per node, and ``p_rewiring`` is the probability of node rewiring.
- The Barabási–Albert preferential attachment graph [BA99] addresses real-world networks’ growth and scale-free nature. Some nodes accumulate connections at a higher rate than others, following a “rich get richer” mechanism. Create it by calling ``nx.barabasi_albert_graph(nodes, edges_per_node)``, where ``nodes`` is the graph size and ``edges_per_node`` is the number of connections a newly arrived node makes.

Together, these graph models offer insights into how social ties form and evolve, the importance of network topology in information dissemination and influence, and the resilience of social systems to changes or attacks.

Functions Returning Graphs

Some NetworkX functions return static graphs of historical importance, such as the “Karate Club” (in Figure 7.1 on p. 128) or the network representation of Victor Hugo’s famous novel *Les Misérables*. Such graphs are often used as a dataset in network analysis and graph theory to illustrate concepts, algorithms, and methods for studying complex networks. You can instantiate them by calling ``nx.social.karate_club_graph()`` and ``nx.social.les_miserables_graph()``.

Reading and Writing Graphs from and to Files

NetworkX recognizes several external representations implemented as a family of ``nx.read_FMT()`` and ``nx.write_FMT()`` functions. Here, ``FMT`` is the file format name:

- ``adjlist`` or ``edgelist`` for CSV-stored adjacency matrices and edge lists,
- ``graphml`` for GraphML (an XML-based Graph Markup Language),
- ``gexf`` for GEXF (Graph Exchange XML Format),
- ``gml`` for GML (Graph Modeling Language),
- DL (the native format of UCINET software [BEF02]), and
- ``pajek`` for NET (the native format of Pajek software [MB16]).

The formats are often not fully compatible. For example, CSV-based adjacency matrices and GML do not support node and edge attributes but are naturally more compact. Below is an example of a GraphML-to-GML converter.

```
name = "some_network"
G = read_graphml(f"{name}.graphml")
write_gml(G, f"{name}.gml")
```

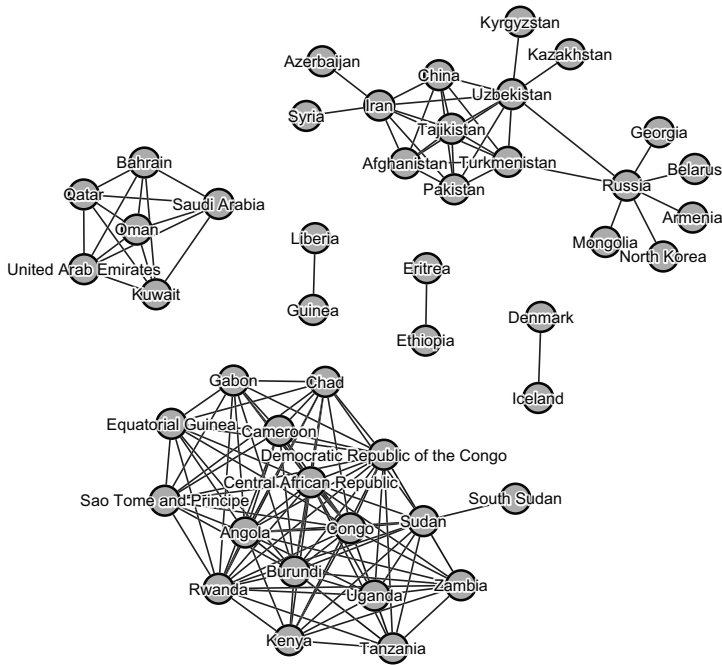
You do not need to open and close the files; NetworkX takes care of them.

The next example illustrates how you can combine the power of NetworkX and Pandas to convert a complex dataset into a complex network. The Correlates of War Formal Alliance dataset [Gib13]² identifies formal alliances between at least two states: defense pacts, neutrality or non-aggression treaties, or entente agreements. The dyadic (state-to-state) dataset can be represented as a graph with the states as the nodes and their mutual pacts as edges. The source data is provided in a CSV file with nineteen columns, making it unsuitable for ``nx.read_edgelist()``. Pandas comes to the rescue: You can read the table with ``pd.read_csv()``, designate the source and destination node columns ("`state_name1`" and "`state_name2`"), and build a network from the edge list with ``nx.from_pandas_edgelist()``. The code in the fragment fragment filters the dataset to keep only those alliances formed in the 21st century.

```
df = pd.read_csv("data/alliance_v4.1_by_dyad.csv")
df20xx = df.query("dyad_st_year >= 2000")
G = nx.from_pandas_edgelist(df20xx, "state_name1", "state_name2")
nx.write_graphml(G, "alliances-20xx.graphml")
```

Figure 15.1 shows the resulting network. The next Subsection explains how to access its nodes, edges, and their attributes.

²<https://correlatesofwar.org/data-sets/formal-alliances/>

**FIGURE 15.1**

The Correlates of War Formal Alliance dataset in the 21st century represented as a complex network.

15.2.2 Accessing Nodes, Edges, and Their Attributes

A graph `G` has `G.number_of_nodes()` nodes and `G.number_of_edges()` edges.

You can access lists of graph nodes and edges via the `G.nodes` and `G.edges` accessors, respectively. As an oddity, the accessors may be called as functions `G.nodes()` and `G.edges()`, and in the latter case, you can pass a `data=True` parameter to them to obtain node and edge attributes. The node/edge lists are represented as immutable (not directly modifiable) but iterable objects `NodeView`, `NodeDataView`, `EdgeView`, and `EdgeDataView`.

NetworkX stores node and edge attributes in dictionaries. You can view and modify them directly or by calling functions.

```
karate = nx.karate_club_graph()
# Access an existing attribute
print(karate.nodes[0]["club"])
#> Mr. Hi
# Create a new attribute
karate.nodes[0]["observed"] = True
```

```
karate.edges(10, data=True)
# Observe existing attributes
print(karate.edges(10, data=True))
#> [(10, 0, {'weight': 2}), ..., (10, 5, {'weight': 3})]
```

To update or obtain the attributes of all nodes or edges, use functions `nx.set_node_attributes()` and `nx.get_node_attributes()` (for nodes) and `nx.set_edge_attributes()` and `nx.get_edge_attributes()` (for edges). For example, the following code fragment calculates edge betweenness centralities for all edges (not elaborated in this book), records them as edge attributes, and reports the value from the edge connecting nodes 0 and 1.

```
ebc = nx.edge_betweenness centrality(karate)
nx.set_edge_attributes(karate, ebc, "betweenness")
print(nx.get_edge_attributes(karate, "betweenness")[(0, 1)])
#> 0.025252525252525245
```

15.3 Other Network Representations

NetworkX can convert a graph G to an adjacency matrix A (Section 7.2 on p. 132) through a call to the `nx.adjacency_matrix(G)` function. The function returns an array, yet it is not the array you would expect. Since a typical complex network is sparse (has relatively few edges), storing it in a full-fledged square array is highly inefficient. Instead, NetworkX uses a compressed sparse row (CSR) array from the NumPy ecosystem. A CSR array contains only non-zero elements and their indexes. It is inefficient in its way, requiring more storage per element, but it works for sparse graphs. For further processing, you must convert a CSR array to a regular 2D array; function `csr_matrix.toarray()` returns a dense NumPy array:

```
adj = nx.adjacency_matrix(karate).toarray()
#> array([[0, 4, 5, ..., 2, 0, 0],
#>        [4, 0, 6, ..., 0, 0, 0],
#>        [5, 6, 0, ..., 0, 2, 0],
#>        ...,
#>        [2, 0, 0, ..., 0, 4, 4],
#>        [0, 0, 2, ..., 4, 0, 5],
#>        [0, 0, 0, ..., 4, 5, 0]])
```

The numbers in the matrix represent the weights of the connections. Since the graph is undirected, the matrix is symmetric.

```
print((adj.T==adj).all())
#> True
```

If you aim to integrate your network data into a Pandas pipeline, you may want to transform a graph to a Pandas dataframe with ``nx.to_pandas_adjacency(G)``. The dataframe has the same elements but with row and column indexes.

A more impressive cross-representation tool converts a graph to a row-wise Pandas dataframe with one edge per row. In addition to the edge's source and destination, the dataframe contains all edge attributes, including the weight.

```

nx.to_pandas_edgelist(karate)
#>      source  target  weight  betw
#> 0         0       1       4  0.025253
#> 1         0       2       5  0.077788
#> 2         0       3       3  0.020499
#> 3         0       4       3  0.052288
#> 4         0       5       3  0.078134
#> ..      ...      ...      ...      ...
#> 73        30       32       3  0.017053
#> 74        30       33       3  0.026814
#> 75        31       32       4  0.041434
#> 76        31       33       4  0.053394
#> 77        32       33       5  0.008225
#>
#> [78 rows x 5 columns]

```

Now, you can use the attributes in regressions and classification tasks.

15.4 Exploring Network Structure

Large-scale complex network structure deals with the hierarchical decomposition of the network graph into constituents.

15.4.1 Connected Components and Subgraphs

Most network analysis algorithms expect the network graph to be connected—in other words, to be a connected component. Disconnected networks require special attention, such as extracting the giant connected component (GCC) or at least acknowledging their disconnectivity.

Connectedness of a graph (the existence of a path connecting two nodes for any pair of nodes) depends on its directedness. In an undirected network, a path from N1 to N2 guarantees the existence of a reverse path, but not so in directed networks (revisit p. 131 for the definitions). Functions ``nx.connected_components()``, ``nx.strongly_connected_components()``, and ``nx.weakly_connected_components()`` are NetworkX's tools for identifying connected components. The former one works only with simple graphs,

the latter two—with digraphs. All three functions are “lazy” generators and produce one component as a collection³ of nodes per call.

To test the functions, let’s do something insane: combine the Karate Club network and the network of *Les Misérables*. They do not share any actors and will remain disconnected. Then, let’s make the new network directional and add a directional connecting edge between Anzelma of *Les Misérables* and the club member number 0.

```
les_miserables = nx.social.les_miserables_graph()
karate.add_edges_from(les_miserables.edges())
di_karate = nx.DiGraph(karate)
di_karate.add_edge("Anzelma", 0)
```

The new edge adds weak connectivity to the graph. Connectedness analysis produces three component lists, confirming that the network with the new edge is weakly connected but not strongly connected.

```
comps1 = list(nx.connected_components(karate))
comps2 = list(nx.strongly_connected_components(di_karate))
comps3 = list(nx.weakly_connected_components(di_karate))
print(len(comps1), len(comps2), len(comps3))
#> 2 2 1
```

Once you identify the components’ composition, you can extract the sub-graphs with the namesake function `nx.subgraph(G, nodes)`.

```
print(nx.subgraph(di_karate, comps1[0]).nodes)
#> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
#> 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
#> 33]
print(nx.subgraph(di_karate, comps1[1]).nodes)
#> ['Napoleon ', 'Myriel ', 'MlleBaptistine ', 'MmeMagloire ',
#> 'CountessDeLo ', 'Geborand ', 'Champtercier ', 'Cravatte ',
#> 'Count ', 'OldMan ', 'Valjean ', 'Labarre ', 'Marguerite ',
#> ..., 'BaronessT ', 'MlleVaubois ', 'MotherPlutarch ']
```

The analysis successfully separated the original networks. But which of them forms the GCC? No special functions are needed to find out. Prime the components’, sort it by the components’ length in the ascending (default) order, and take the last element. Its size is the largest. It describes the largest component. In the natural (not experimental) setting, the largest component will be much larger than all other components.

```
# Generator!
components = nx.strongly_connected_components(di_karate)
gcc_names = sorted(components, key=len)[-1]
gcc = nx.subgraph(di_karate, gcc_names)
```

³The collection is known as a set. The book introduces sets in [Section 16.1](#) on p. 298.

15.4.2 Projecting Bipartite Graphs

Bipartite graphs are a subclass of graphs consisting of two groups of nodes such that no two nodes in the same group share an edge⁴. While seemingly insignificant, the restriction on connectivity strongly affects the graph topology and all associated network measures. For example, the density of a bipartite graph is, on average, only half the density of a unipartite graph. When dealing with a bipartite graph, you should use specialized bi-algorithms (such as `nx.bipartite.degrees()` and `nx.bipartite.density()`) or project one part onto the other.

The first step towards projection is to determine if the graph is bipartite (we use the Davis Southern Women Event network from [Figure 7.6](#) on p. 142 as an example):

```
davis = nx.read_gml("data/davis.gml")
print(nx.is_bipartite(davis))
#> True
```

Once confirmed, you can separate the nodes into two groups (sets):

```
davis_s = nx.bipartite.sets(davis)
print(davis_s)
#> ({'Verne Sanderson', 'Sylvia Avondale', 'Eleanor Nye',
#> 'Katherina Rogers', 'Olivia Carleton', 'Pearl Oglethorpe',
#> 'Evelyn Jefferson', 'Ruth DeSand', 'Myra Liddel',
#> 'Flora Price', 'Helen Lloyd', 'Charlotte McDowd',
#> 'Frances Anderson', 'Dorothy Murchison', 'Nora Fayette',
#> 'Brenda Rogers', 'Theresa Anderson', 'Laura Mandeville'},
#> {'E8', 'E7', 'E13', 'E4', 'E9', 'E1', 'E12', 'E2', 'E14',
#> 'E3', 'E11', 'E5', 'E10', 'E6'})
```

Projection is the last step of the process. You can project a bipartite graph in two ways by keeping one group of nodes and eliminating the other. In this example, you can get a unipartite network of women (without events) or a unipartite network of events (without women). To automate the procedure, identify the group of nodes you are interested in (say, the group of events). Since one of the events is called “E1,” you can select the group of event nodes by looking for “E1” in each group and finally execute the projection algorithm:

```
events = davis_s[0] if ("E1" in davis_s[0]) else davis_s[1]
E = nx.bipartite.weighted_projected_graph(davis, events)
```

The projection result is graph `E`, whose nodes are `events`. Two event nodes are connected with an edge if at least one woman attended both. The graph is weighted; the weight of an edge is the number of co-attending women. To rephrase, `E` is a unipartite weighted undirected graph you can analyze

⁴Tripartite graphs exist, too. They may denote the relationships between organizations, employees, and customers.

like any other such graph. [Section 15.6](#) on p. 295 explains and presents the visualization of the graph.

15.4.3 Cliques, Cores, and Communities

If components are the continents of a complex network, then cliques, cores and network communities are its political blocks, unions, and countries.

Cliques and Cores

Cliques ([Subsection 7.4.1](#) on p. 138) are the easiest structural elements of a complex network to understand. All nodes are connected in a clique, forming a fully connected graph. A monad—a node—is a clique. A dyad—a pair of adjacent nodes—is a clique. A triad—a group of three pairwise adjacent nodes—is a clique, too. Any fully connected graph is naturally a clique.

Cliques are also some of the most challenging structural elements of a complex network to compute because the number of cliques in a graph can be exponential in the number of nodes. That is why the function ``nx.clique.find_cliques(G)`` is implemented as a generator and produces one clique per call. You may still externalize the generator as a list.

The code below explores the distribution of all clique sizes and maximal clique (p. 138) sizes in the Karate Club network with the functions ``nx.enumerate_all_cliques(G)`` and ``nx.find_cliques(G)`` and Pandas.

```
pd_all_cliques = pd.Series(nx.enumerate_all_cliques(karate))
pd_max_cliques = pd.Series(nx.find_cliques(karate))
clique_all_sizes = pd_all_cliques.str.len().value_counts()
clique_max_sizes = pd_max_cliques.str.len().value_counts()
sizes = pd.DataFrame({"all": clique_all_sizes,
                      "max": clique_max_sizes})
print(sizes.fillna(0).astype(int))
#>      all  max
#> 1     34    0
#> 2     78   11
#> 3     45   21
#> 4     11    2
#> 5      2    2
```

Dyads and, to some extent, triads are the most popular cliques. The operation you just performed is called a motif census: counting specific network configurations (usually to compare two networks for structural similarities). Cliques themselves are used to decompose a graph into dense(r) neighborhoods. You can treat those neighborhoods as categories for classification.

A graph core (a k -core) is an artifact of the core-periphery analysis. Cores are philosophically complementary to cliques: the latter look for dense subgraphs, and the former “peel off” the sparse, peripheral subgraph shells. The k^{th} core of ``G`` is obtained by progressively removing all nodes with the degree

of k until no such nodes are left. Each next core has a higher density than the previous one and is a subset of it. (Cliques do not have this enclosure property.)

The function `nx.k_core(G, k)` returns the k^{th} core subgraph (nodes and edges). The function returns the densest core unless you provide the optional parameter `k`.

Community Detection

Cliques and cores are strictly defined mathematical (graph theoretical) concepts. As such, they may be hard to find or interpret in real-life complex networks. For example, removing any edge (a routine event in social networks!) destroys an enclosing clique but rarely and barely affects the clique members.

Network community detection is a more relaxed, non-deterministic procedure that decomposes the graph into non-intersecting communities of various sizes by maximizing the partition's modularity. Modularity emphasizes intra-community edges over inter-community edges. Higher modularity implies more of the former and fewer of the latter.

NetworkX (version 3.2.1 and above) has a built-in function for modularity-based community detection, `nx.community.louvain_communities(G)`, eliminating the need for third-party modules. The function is customizable but works well with the default values of all parameters. It returns a list of node groups that could be used for subgraph extraction and node labeling. The reporting function `nx.community.modularity(G, partition)` evaluates the modularity of the `partition`. You can convert the partition into a dictionary of nodes/partition identifiers and then further into a Pandas series.

```
parts = nx.community.louvain_communities(karate)
modularity = nx.community.modularity(karate, parts)
print(modularity)
#> 0.4276718952043627
parts_dict = {n: part for part, nodes in enumerate(parts)
              for n in nodes}
print(pd.Series(parts_dict, name="node_id"))
#> 1      0
#> 2      0
#> 3      0
#> 7      0
#> 12     0
#> 13     0
#> 0      1
#> 4      1
#> ...
#> 27     3
#> 28     3
#> 31     3
#> Name: node_id, dtype: int64
```

Bear in mind that due to the Louvain algorithm's probabilistic nature, the actual partition and the resulting modularity may differ from the previously obtained results. If you need to obtain a trustworthy graph partition, you should at least repeat the procedure several times.

15.4.4 Density and Clustering Coefficient

Individual nodes and their neighborhoods form the network's lowest structural hierarchy level. They are quantified by their local density ([Subsection 7.3.5](#) on p. 137): the number of actual edges between the neighbors divided by the number of possible edges. Known as the clustering coefficient, the local density is calculated by function `nx.clustering(G)` and returns a dictionary of nodes, naturally convertible into a Pandas series.

```
cc = nx.clustering(karate)
cc_ser = pd.Series(cc, name="clustering")
print(cc_ser)
#> 0      0.150000
#> 1      0.333333
#> ...
#> 32     0.196970
#> 33     0.110294
#> Name: clustering, dtype: float64
```

The clustering coefficient is often a deceiving measure of a node's importance and should be taken with a grain of salt. For example, it always equals 1 for the nodes with only one incident edge.

As another nod to the concept of the neighborhood, function `nx.neighbors(G, node)` returns a generator of the `node`'s network neighbors.

Now that we have zoomed in on the level of individual nodes let us zoom out and mention the global network density, which is the ratio of existing edges to the possible number of edges in a fully connected graph with the same nodes. The function `nx.density(karate)` returns a modest value of 0.139 (13.9%), confirming that a real-world social network is typically sparse.

15.5 Calculating Centralities

Calculating various network centralities: closeness centrality, degree centrality, betweenness centrality, and eigenvector centrality—is as simple as calling the namesake functions conveniently supplied by NetworkX:

```
degree = nx.degree_centrality(karate)
closeness = nx.closeness_centrality(karate)
```

TABLE 15.1
Correlations between network measures in the karate club network

	degree	closeness	betweenness	eigenvec	clustering
degree	1.000000	0.771591	0.914643	0.917255	-0.515168
closeness	0.771591	1.000000	0.717945	0.904615	-0.612168
betweenness	0.914643	0.717945	1.000000	0.803213	-0.480666
eigenvec	0.917255	0.904615	0.803213	1.000000	-0.454318
clustering	-0.515168	-0.612168	-0.480666	-0.454318	1.000000

```
betweenness = nx.betweenness_centrality(karate)
eigenvec = nx.eigenvector_centrality(karate)
```

Each function returns a dictionary of nodes. You can convert each dictionary into a Pandas series or, preferably, all four dictionaries into a Pandas dataframe.

```
measures = pd.DataFrame({"degree": degree, "closeness": closeness,
                        "betweenness": betweenness, "eigenvec": eigenvec})
```

The dataframe may include other node-level parameters, such as the clustering coefficient and clique, core, shell, and network community identifiers. In such a state, it becomes suitable for running regression and classification tasks, in which any of the network measures can act as a dependent of an independent variable.

However, be advised that the centralities are not usually mutually independent. [Table 15.1](#) (with clustering coefficient included) shows that the correlation between the centralities is too strong to be ignored. Applying principal component analysis (PCA) or factor analysis may alleviate the effect of multicollinearity, but both methods are outside the book’s scope.

15.6 Network Visualization

The most recent version of NetworkX achieved significant progress in network visualization. You can draw a network in one step by calling `nx.draw()` or gradually, layer by layer, as shown below.

The first two steps concern data preparation and do not involve actual drawing. Use function `nx.spring_layout(G)` to calculate the positions of the nodes on the output canvas. In addition to the spring layout, NetworkX supports several other presentation layouts: circular, Fruchterman–Reingold, Kamada–Kawai, spiral, and more. Choose the most aesthetically appropriate for your graph.

The edge attribute `"weight"` will control the edges' width. Extract it from the graph and organize it in a list of numbers. Depending on the range of weights in your graph, you might need to scale these values to ensure they are visually distinguishable and aesthetically pleasing.

The subsequent three functions draw edges, nodes, and labels, respectively. They need to know the graph, the nodes' positions, and the edges' weights. All other information (edge color, edge transparency, node size, node color, label font size, and label font color) is optional. The order of the calls is essential: edges should be drawn first, followed by the nodes, followed by the labels on the nodes.

```
pos = nx.spring_layout(E)
weights = list(nx.get_edge_attributes(E, "weight").values())
nx.draw_networkx_edges(E, pos, edge_color="gray", alpha=0.75,
                      width=weights)
nx.draw_networkx_nodes(E, pos, node_size=700, node_color="black")
nx.draw_networkx_labels(E, pos, font_size=13, font_color="white")
```

The code output is shown in [Figure 15.2](#). You are encouraged to experiment with alternative drawing layouts, colors, and styles.

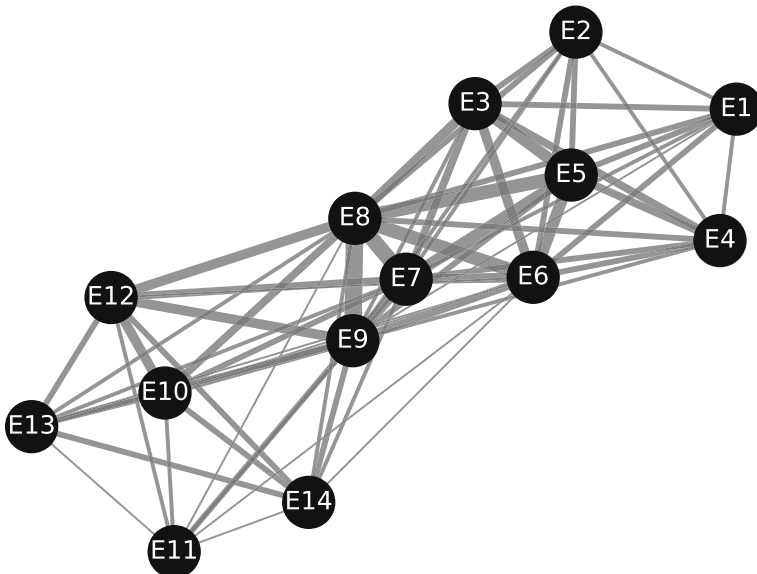


FIGURE 15.2

Weighted projected Davis Southern women event network (the event view); cf. [Figure 7.7](#) on p. 143.

NetworkX's primary strength lies in network analysis rather than advanced visualization. Additional tools like Graphviz or Gephi might be more appropriate for more complex or aesthetically pleasing visualizations.

15.7 What's Next

It is infeasible to cover all aspects of complex network analysis in one chapter. If you are interested in the field, refer to [Zin18] and other specialized books. Meanwhile, the final chapter will introduce you to the foundations of natural language processing and show you how to treat text as a potential source of statistical data.

15.8 Programming Exercises

Exercise 23. Write a function `file_format(fname, width)` that opens the file `fname` and, at each call, yields the next line from the file truncated to the first `width` characters. The default value of `width` is 80.

Exercise 24. Annotate Zachary's Karate Club network nodes by assigning to them the attribute `"max_clique"`. The attribute's value is 1 for the nodes belonging to the network's largest clique and 0 otherwise. Draw the network and color the nodes based on the attribute's value.

Exercise 25. Using linear (OLS) regression and Zachary's Karate Club network, determine which three of the four centralities mentioned in [Section 15.5](#) on p. 294 are the best predictors for the remaining fourth centrality. Use R-squared as the quality-of-prediction measure.

16

Sets. Text as Data

The printed material collection of the Library of Congress has about 10 terabytes (10,000 gigabytes) of text. Nobody (even Google!) knows how much text is on the Internet, but according to some estimations, the global data-sphere will grow to 175 zettabytes (175,000,000 gigabytes) by 2025. Not all of those zettabytes are text: conservatively, text constitutes only 1% of them, ≈ 2 zettabytes, still an impressive number.

Text is an invaluable source of data for a social science researcher. This chapter explains how to use Natural Language Processing (NLP) to work with text as data. But first, here is a short and needed introduction to Python sets.

16.1 Sets

A set is a mutable collection of unique, immutable Python elements. A set is heterogeneous: you can mix and match numbers of all kinds, strings, tuples, booleans, and `None` values—but not lists, dictionaries, or other sets.

16.1.1 Set Properties and Operations

You can construct a set by calling the `set(X)` constructor, which converts `X` (a list, a tuple, or any other sequence) into a set. Alternatively, you can enclose a list of set items in a pair of curly braces, similar to a literal dictionary declaration but without values. A set can be empty—created via `set()`. An empty set contains no elements and is zero size. Here are some set examples:

```
print(set()) # An empty set
#> set()
# Text as a set of words, numbers, and punctuation
text = {"Mary", "had", 100, "little", "lambs", "."}
print(text)
#> {'.', 100, 'had', 'lambs', 'Mary', 'little'}
# A set of all decimal digits
import string
print(set(string.digits))
#> {'4', '3', '7', '9', '8', '0', '1', '5', '6', '2'}
```

Note that the order of items in a set is undefined and unimportant. What matters is whether an item is in the set or not in the set.

A set is mutable: you can add and remove items to and from it. An attempt to remove a non-existing item causes an error. Conversely, when you add an item already in the set, the operation is silently ignored because sets do not allow duplicates.

```
text.add("hundred")
print(text)
#> {'.', 100, 'hundred', 'had', 'lambs', 'Mary', 'little'}
text.add("hundred") # Add another "hundred"!
print(text)
#> {'.', 100, 'hundred', 'had', 'lambs', 'Mary', 'little'}
text.remove(100)
print(text)
#> {'.', 'hundred', 'had', 'lambs', 'Mary', 'little'}
```

You can check if an element is in a set using the namesake operators: ``in`` and ``not in``. The operators are boolean; they evaluate to ``True`` or ``False`` and can be used in conditional statements.

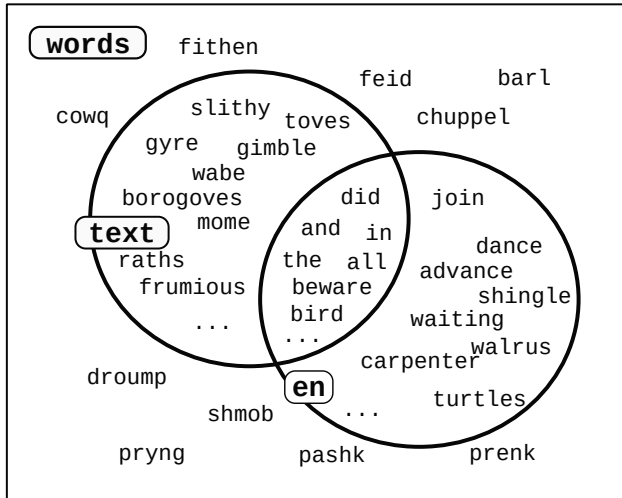
```
print("Mary" in text)
#> True
print("little" not in text)
#> False
print(100 not in text)
#> True
if "Mary" in text:
    print("This text is about Mary.")
#> This text is about Mary.
```

Python sets support all primary set-theoretical operations: union, intersection, and difference. [Figure 16.1](#) illustrates the operations using a Venn diagram. The diagram shows a set of words (“words”), some of which are taken from Lewis Carroll’s *Jabberwocky* (subset “text”), some being English words (subset “en”), and some being random combinations of letters (outside of the two circles).

A union of sets ``s1`` and ``s2`` is a set whose elements belong to ``s1`` or to ``s2`` (or, naturally, to both). The union of ``text`` and ``en`` is denoted ``(text | en)`` and is a set of words that are either proper English words or words used by Carroll.

An intersection of sets ``s1`` and ``s2`` is a set whose elements belong to both ``s1`` and ``s2``. The intersection of ``text`` and ``en`` is denoted ``(text & en)`` and is a set of proper English words used by Carroll (such as “did” and “bird”).

A set difference between ``s1`` and ``s2`` is a set whose elements belong to ``s1`` but not ``s2``. The difference between ``text`` and ``en`` is denoted ``(text`

**FIGURE 16.1**

Venn diagram illustrating set operations on words.

- en)`; it is a set of *Jabberwocky* words that are not English words (such as “raths” and “toves”). Unlike the union and intersection, the set difference is not symmetric:  $s_1 - s_2 \neq s_2 - s_1$ . The difference between `en` and `text` ($\text{en} - \text{text}$) is a set of English words not used in *Jabberwocky* (such as “join” and “turtles”).

Given the basic operations, you can efficiently compute the “pure garbage” word set: $\text{words} - \text{text} - \text{en}$.

16.1.2 Uses of Sets

Stop word elimination is one of the early-stage NLP activities (See [Subsection 16.3.3](#) on p. 305). In the gist, you want to know if a word is on the list of stop words, either a standard one (e.g., imported from `nltk.corpus.stopwords`) or custom-made. The `in` operator is used to check the word membership on the list. You may need to search for every word of the processed text.

```
import nltk
stopwords = nltk.corpus.stopwords.words("english")
for word in text:
    if word not in stopwords:
        do_something_with(word)
```

Checking word membership in a long list takes significant time, which gets longer as the list gets longer. Finding “peace” (not a stop word) in the

“official” 180-item list of English language stop words takes 1.74 microseconds. Consider converting the list to a `set` before performing any lookups:

```
stopwords = set(stopwords)
for word in text:
    if word not in stopwords:
        do_something_with(word)
```

The time is now 0.03 microseconds, almost 60 times faster.

The difference may be life-changing for a large text with hundreds of thousands of words. Is “frumious” an English word¹? Try to find it on the list of 236,000 English words; the operation takes 2,610 microseconds. After converting the list of words to a set of words, the lookup time becomes as fast as 0.038 microseconds, $\approx 70,000$ times faster.

Another popular application of sets is deduplication, removing duplicate items from a list. The `set(X)` function naturally eliminates duplicates from `X`.

```
l = "Whereas recognition of the inherent dignity and of the equal"
    "and inalienable rights of all members of the human family is"
    "the foundation of freedom justice and peace in the world"
    .lower().split()
print(len(l))
#> 31
s = set(l)
print(s)
#> {'inherent', 'is', 'rights', 'foundation', 'freedom',
#> 'whereas', 'justice', 'and', 'dignity', 'all', 'recognition',
#> 'the', 'peace', 'of', 'human', 'in', 'inalienable', 'equal',
#> 'world', 'family', 'members'}
print(len(s))
#> 21
```

The result does not contain duplicates. On the dark side, the original order of words is lost in conversion.

16.2 HTML and BeautifulSoup

We will illustrate the NLP pipeline using the sample HTML fragment introduced in [Subsection 8.1.1](#) on p. 148.

HyperText Markup Language (HTML) is a standard language for marking up (“annotating”) text documents. The first version of HTML was designed in 1991 by Tim Berners-Lee, followed by version 2.0 in 1995, and eventually by

¹It is not.

HTML 5 in 2014. Unlike Python, HTML is not a programming language. It lacks variables, conditional statements, loops, functions, and other programming elements. You cannot write an HTML program.

Instead, HTML provides tags. A tag is a command enclosed in angle brackets ``<`` and ``>``. A tag consists of a start tag and, optionally, a matching end tag. A slash character ``/`` precedes the end tag's name. The tag command affects the content between the start and end tags. The following fragment of the HTML document contains two ``<SPAN>`` start tags (controlling the visual properties of the text span—its style) with the matching ``</SPAN>`` end tags and a self-closing tag ``<BR>`` (a line break). The latter may be written as ``<BR/>`` to emphasize its self-sufficiency. Note that HTML tags are case-insensitive.

```
<span style='text-decoration: underline;'>I am</span> <span
  style='font-weight: bold;'>sad</span> because I feel
  helpless. <br>
```

Tags can have attributes: key/value pairs akin to Python dictionaries. In the fragment above, the first attribute ``style` has the value “text-decoration: underline;” which instructs the browser to underline the text between the start and end tags. In the same fragment, the second attribute `style` has the value “font-weight: bold;” which causes the text to be shown in bold font.`

Web browsers do not display tags. They interpret them as structuring and styling instructions. When you perform natural language processing, you generally want to concentrate on the text between and outside the tags. Python module ``BeautifulSoup`2 provides tools for parsing, editing, and generating HTML documents. The module is called `bs4` (“BeautifulSoup” version 4”). Once imported, the constructor converts an HTML document to a “soup” object. Note that the document can be a string or an open file handle.`

```
from bs4 import BeautifulSoup
with open("data/417765.html") as doc:
    soup = BeautifulSoup(doc)
```

The ``soup` object has nearly 200 attributes and functions, of which we need just one: `soup.text` . The attribute's value is a string that contains the original text stripped of the tags (see Listings on p. 148 and below).`

```
text = soup.text
print(text)
#> Look what I learned in therapy today mommy!, I had no clue I
#> could be my own person... I am upset because I can't speak my
#> mind. I am angry because I can't voice what I want heard. I am
#> sad because I feel helpless. I am hurt because I ruined my
```

²“Beautiful Soup” refers to the Mock Turtle’s song from Lewis Carroll’s *Alice’s Adventures in Wonderland*. The song is about a whimsical and nonsensical soup that contains various ingredients far beyond the ordinary.

#> *life & stuff. that's absolutely amazing!!*

And you are ready to engage with NLTK.

16.3 Natural Language Toolkit (NLTK)

Natural Language Toolkit (NLTK, [BKL09]) is a popular Python library for natural language processing (NLP). While not comprehensive, it suffices for routine tasks, such as text preprocessing, stemming, lemmatization, PoS tagging, extractive summarization, named entity recognition (NER), and sentiment analysis.

16.3.1 Tokenization and Language Detection

Text tokenization: splitting a text into tokens, such as words, numbers, and punctuation—is often inseparable from language detection (see [Subsection 8.1.2](#) on p. 148) because different languages may have different tokenization rules. Let's hypothesize that the document's text is written in English and confirm (or reject) this hypothesis.

Start by importing the “words” corpus: a collection of the most popular words in a chosen language. You can take an extra step and look at two supported corpora: “en” for the standard English language and “en-basic” for the British American Scientific International and Commercial English, an 850-word stripped-down version of English created in 1925 by Ivor Richards and Charles Kay Ogden. At the time of writing, the default NLTK package does not have any other language word corpora.

A corpus is an instance of a `WordListCorpusReader` from the submodule `nltk.corpus.words`. The reader provides a function `words(X)` that returns a list of words and forms in the language `X`. Since you will compare this list with the list of words in the text, convert it to a Python `set` for a faster lookup.

```
import nltk

english_words = set(nltk.corpus.words.words("en")) # 235886
basic_words = set(nltk.corpus.words.words("en-basic")) # 850
```

The function `nltk.word_tokenize(text)` tokenizes the `text` and returns a list of tokens. The function `nltk.casual_tokenize()` works similarly but does not break up contractions such as “that’s.” Function `nltk.wordpunct_tokenize()` keeps all sequential punctuation together (e.g., it returns “!!” as one token, not two). Two more tokenizers, `nltk.sent_tokenize()` and `nltk.line_tokenize()`, partition

the ``text`` into sentences and lines, respectively. You need either ``nltk.word_tokenize(text)`` or ``nltk.casual_tokenize(text)``, depending on how you treat contractions.

The list of tokens contains words, numbers, and punctuation. You can apply the function ``str.isalpha()`` to each token and keep only the tokens for which the predicate is true. The standard Python function ``filter(function, sequence)`` does just this: it applies the ``function`` to each item in the ``sequence`` and constructs a generator of the items that passed the test. Convert the surviving tokens to the lowercase because the lists of English words are in the lowercase. The function ``map(function, sequence)`` is your tool for this job. Finally, construct a set of the words for further set-theoretic operations.

```
tokens = nltk.word_tokenize(text)
words = set(
    map(str.lower,
        filter(str.isalpha,
               tokens))) # 36 words
```

You ended up with three sets of words: ``words``, ``english_words``, and ``basic_words``. To what extent does the set of words from the text intersect with the language sets?

```
print(len(words & english_words) / len(words))
#> 0.9722222222222222
print(len(words & basic_words) / len(words))
#> 0.3055555555555556
```

The answers suggest that the text is in the English (97%) or Basic English (31%) languages but mentions too many non-basic words³.

16.3.2 Part-of-Speech Tagging

The function ``nltk.pos_tag(tokens)`` annotates each token in the list with a part-of-speech tag. The tag is a string that identifies different classes and subclasses of parts of speech, such as VERBs, PRONouns, NOUNs, and prepositions.

```
tokens = nltk.casual_tokenize(text)
pos_tags1 = nltk.pos_tag(tokens)
print(pos_tags1[:10])
#> [('Look', 'VB'), ('what', 'WP'), ('I', 'PRP'),
#> ('learned', 'VBD'), ('in', 'IN'), ('therapy', 'NN'),
#> ('today', 'NN'), ('mommy', 'NN'), ('!', '.'), (',', ', ')]
```

³Quite unexpectedly, the Basic English vocabulary is not a subset of English vocabulary. Due to the British spelling used in the former, the words “harbour,” “behaviour,” “humour,” and “colour” are Basic English but not [American] English.

```
pos_tags2 = nltk.pos_tag(tokens, tagset="universal")
print(pos_tags2[:10])
#> [('Look', 'VERB'), ('what', 'PRON'), ('I', 'PRON'),
#>  ('learned', 'VERB'), ('in', 'ADP'), ('therapy', 'NOUN'),
#>  ('today', 'NOUN'), ('mommy', 'NOUN'), ('!', '.'), (',', '.')]

```

NLTK supports different tagsets, which are selectable via the ``tagset`` optional argument. Some tagsets are more specific or more straightforward to interpret than others. For example, the “universal” tags are self-explanatory, but not those in the default “UPenn” tagset. You can learn about their meaning by querying their names:

```
nltk.help.upenn_tagset("NN")
#> NN: noun, common, singular or mass
#>      common-carrier cabbage knuckle-duster Casino afghan shed
#>      thermostat investment slide humour falloff slick wind
#>      hyena override subhumanity machinist ...
nltk.help.upenn_tagset("PRP$")
#> PRP$: pronoun, possessive
#>      her his mine my our ours their thy your

```

Stop words ([Subsection 8.1.6](#) on p. 152) may affect PoS tagging, but other than that, they usually do not deserve your attention and can now be eliminated.

16.3.3 Stop Word Elimination

NLTK provides a corpus of stop words in several popular natural languages, including English, Russian, Tajik, and Nepali. The list of languages is available by calling ``nltk.corpus.stopwords.fileids()``. An NLTK corpus is structured as a collection of files. The ``fileids()`` are file IDs, one file per language.

Standard stop-word lists are helpful but not perfect. First, they are inconsistently partial: for example, the English list contains words “can” and “wouldn’t,” but not “can’t” or “would.” Second, they are generic and may not address your needs as a researcher in a specific field. For example, if you process a corpus of texts about political parties, both “political” and “party” will be abundant. Fortunately, you can easily modify the list (or, rather, a set) by adding new stop words and removing the words you do not consider stop words.

```
1 stop_set = set(nltk.corpus.stopwords.words("english"))
2 extra_stops = {"could", "can't", "that's", "would", "party"}
3 stop_set |= extra_stops
4 clean_text = [w for w in tokens if w.lower() not in stop_set]
5 print(clean_text)
6 #> ['Look', 'learned', 'therapy', 'today', 'mommy', '!', ',',
7 #>  'clue', 'person', '...', 'upset', 'speak', 'mind', '.',
8 #>  'angry', 'voice', 'want', 'heard', '.', 'sad', 'feel',

```

```

9 #> 'helpless ', '. ', 'hurt ', 'ruined ', 'life ', '&', 'stuff ',
10 #> '. ', 'absolutely ', 'amazing ', '!', '!' ]

```

Once you adjust the set, traverse the list of the original tokens and discard those found in the set of stop words.

Note the augmented operator ``|=`` on line 3. It combines the set union ``|`` and the assignment operator ``=``. The expression is an abbreviation for ``stop_set = stop_set | extra_stops`` and is considered “syntactic sugar”: a feature that makes the code easier to read but does not change its meaning.

16.3.4 Stemming and Lemmatization

Stemming and lemmatization are often the last two essential text preprocessing steps.

Stemmers lower inflection in words to their root form. They identify and remove endings and suffixes such as “-s,” “-ing,” “-ly,” and “-ed.” The process of stemming is defined informally and is implementation-specific. Some stemmers (e.g., the Lancaster stemmer) are “mild” and remove relatively few word parts; others (e.g., the Porter stemmer) are more “aggressive” and often over-mutilate words. Aggressive stemmers produce fewer root forms with higher frequencies at the expense of sacrificing word flavors, while mild stemmers do the opposite.

The Lancaster and Porter stemmers are available in NLTK. Once instantiated, they provide the method ``stem(X)``, which stems the previously tokenized word ``X``. Compare the stemming results:

```

ps = nltk.PorterStemmer()
ls = nltk.LancasterStemmer()

print(" ".join(ps.stem(w) for w in clean_text))
#> look learn therapi today mommi ! , clue person ... upset speak
#> mind . angri voic want heard . sad feel helpless . hurt ruin
#> life & stuff . absolut amaz ! !

print(" ".join(ls.stem(w) for w in clean_text))
#> look learn therapy today mommy ! , clu person ... upset speak
#> mind . angry voic want heard . sad feel helpless . hurt ruin
#> lif & stuff . absolv amaz ! !

```

Your analysis’s specific needs would determine the choice between these stemmers. A more aggressive stemmer might be useful in contexts where the exact form of a word is less important than its root meaning. On the other hand, a less aggressive stemmer might be preferred for problems where preserving some part of the word’s original form is beneficial.

Sadly (but expectedly), the root forms produced by stemmers are often not valid word forms and are not in the ``nltk.corpus.words`` corpus⁴. A

⁴They are also hard to comprehend because they are intended to be used as statistical variables rather than as reading matter.

lemmatizer (e.g., the WordNet lemmatizer) guarantees to reduce words to lemmas: dictionary root forms.

```
wnl = nltk.WordNetLemmatizer()
print(" ".join(wnl.lemmatize(w) for w in clean_text))
#> Look learned therapy today mommy ! , clue person ... upset
#> speak mind . angry voice want heard . sad feel helpless .
#> hurt ruined life & stuff . absolutely amazing ! !
```

The WordNet lemmatizer works much better if you supply part-of-speech tags. The tags generated by ``nltk.pos_tag()``` (Subsection 16.3.2 on p. 304) differ from those expected by ``wnl.lemmatize()```. The former tags are “a” for an adjective, “v” for a verb, “n” for a noun, and “r” for an adverb. (The remaining parts of speech, such as conjunctions and pronouns, are immutable and cannot be lemmatized.) You can translate the ``nltk.pos_tag()``` tags to ``wnl.lemmatize()``` tags through dictionary lookups based on the first letter of the ``nltk.pos_tag()``` tags.

```
# The dictionary for tag translations. Treat unknown tags as nouns
pos_translate = {"J" : "a", "V" : "v", "N" : "n", "R" : "r"}
words = [wnl.lemmatize(w, pos_translate.get(pos[0], "n")).lower()
          for w, pos in pos_tags1]
# Eliminate stopwords
print(" ".join(w for w in words if w not in stop_set))
#> look learn therapy today mommy ! , clue person ... upset
#> speak mind . angry voice want heard . sad feel helpless .
#> hurt ruin life & stuff . absolutely amazing ! !
```

The text preprocessing phase results in a list or a set of normalized tokens: converted to the standard case, stemmed or lemmatized, and stripped off of stopwords. You can use them as input to higher-level NLP tasks, such as summarization, named entity recognition, sentiment analysis, and topic modeling.

16.4 Extractive Summarization

Extractive summarization is relatively easy to implement in “plain” Python without sophisticated additional modules. Even though the code listing below uses the ``collections.Counter``` object to count lemmas and the ``itertools.chain``` object to combine several lists into one list, their functionality can be easily recreated.

In the spirit of Subsection 8.3.1 on p. 156, let’s perform an extractive summarization of the text of that chapter, subject to the same disclaimer: “Due to the ongoing editing process, the text quoted in this section may differ from the original text.”

The code in the first ten lines of the suggested solution imports the required modules and reads the text from the file. The code in lines 10–21 normalizes the text:

- Tokenizes it two ways (into sentences and words),
- Lemmatizes the words, and
- Eliminates stopwords, as explained previously in this chapter.

Listing 16.1

extractive.py

```

1  # Extractive summarization
2  from itertools import chain
3  from collections import Counter
4  import nltk
5
6  # Read the text
7  with open("data/chapter.txt") as infile:
8      text = infile.read().replace("\n", " ")
9
10 # Tokenize and lemmatize
11 sents = nltk.sent_tokenize(text)
12 wnl = nltk.WordNetLemmatizer()
13 all_lemmas = [[wnl.lemmatize(w.lower())
14                 for w in nltk.casual_tokenize(sent)
15                 if w.isalpha()] for sent in sents]
16
17 # Eliminate stopwords
18 clean_lemmas = [[l for l in lemmas if l not in stop_set]
19                  for lemmas in all_lemmas]
```

The second part of the procedure counts the frequencies of the lemmas and arranges them in a dictionary. We assume that frequently occurring lemmas refer to essential concepts in the text.

```

21 # Count the lemmas
22 counts = Counter(chain.from_iterable(clean_lemmas))
23 scores = dict(counts.most_common())
```

Finally, we calculate scores for each sentence by adding the counts for each lemma. The sentences with the top scores (say, top five) emerge as the summary.

```

25 # Calculate scores
26 contribs = [sum(scores.get(w, 0) for w in clean_sent)
27              for clean_sent in clean_lemmas]
28 threshold = sorted(contribs)[-5]
```

```

29
30 # Select the top-scored sentences
31 for score, s in zip(contribs, sents):
32     if score >= threshold:
33         print(s, end=" ")
34
35 print()

```

Frequency scoring is not the only way to summarize a text. Other methods are based on TextRank/PageRank, latent semantic analysis, Google T5, and other technologies and algorithms. They are outside of this book's scope, and so are abstractive summarization techniques. (Section 8.3 on p. 156 will remind you of the difference between the two summarization mechanisms.)

16.5 Named Entity Recognition

Despite the broad use of NLTK, especially for educational purposes, (NER) is much better handled by the spaCy module.

After installing spaCy, you must download a language model. At the time of writing, spaCy supports 24 languages⁵, including English, Chinese, Spanish, and Russian. English language support includes four free core models:

1. `en\_core\_web\_sm`: the “small” model, 12 MB;
2. `en\_core\_web\_md`: the “medium” model, 40 MB;
3. `en\_core\_web\_lg`: the “large” model, 560 MB;
4. `en\_core\_web\_trf`: the transformer pipeline, 436 MB.

Each model provides a PoS tagger, a sentence tokenizer, a named entity recognizer, a lemmatizer, and other tools.

```

import spacy
spacy.cli.download("en_core_web_sm") # Do just once

```

Once a model is downloaded, it is available to your applications. You can load it and construct a `spacy.lang.en.English` object from the text to be analyzed. The text analysis results are immediately available through the accessor `.ents` and other accessors:

```

import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp(text) # The text of Chapter 8
for entity in doc.ents:

```

⁵<https://spacy.io/models/>

```

    print(entity.text, entity.label_)
#> NLP ORG
#> English LANGUAGE
#> Chinese NORP
#> NLP ORG
#> the 1950s DATE
#> the mid-2000s DATE
#> Google Translate ORG
#> ...
#> the Declaration of Independence of the USA WORK_OF_ART
#> 1776 DATE
#> 363 CARDINAL
#> ...
#> the King and Duchess ORG
#> Alice PERSON
#> Cat LOC
#> Dormouse ORG
#> Queen PERSON
#> the Mock Turtle ORG
#> Chapter 5 LAW

```

The [somewhat questionable] quality of the output reflects the complexity of the NER task, which is at least partially compensated by the model's ease of use. The “large” model performs slightly better, recognizing “Dormouse” as a PERSON and “the Declaration of Independence of the USA” as a LAW (but ignoring “the King and Duchess” and, in a grotesque way, treating “`<span>X</span>`” as a PERSON).

You may ignore most of the named entities' categories: e.g., report only “PERSON,” “ORG[anization],” and “NORP” (“Nationalities or Religious or Political groups”)—but spaCy always extracts all named entities.

16.6 Sentiment Analysis

VADER (Valence Aware Dictionary and sEntiment Reasoner) is an NLTK way of performing sentiment analysis⁶. It has been pre-trained to recognize sentiment level in short text fragments—ideally the size of X (former Twitter) messages.

VADER does not require text preprocessing other than removing URLs and partitioning text into shorter fragments, if necessary. Once you create a VADER instance via `SentimentIntensityAnalyzer()`, you can immediately start the analysis by calling the `polarity_scores(text)` function. The function takes `text` as the parameter and returns a dictionary of four polarity

⁶VADER itself is a standalone sentiment analysis tool that NLTK provides access to.

scores: “neg”[ative], “pos”[itive], “neu”[tral], and “compound.” The first three scores always add up to one, but the fourth is independent.

Let’s use VADER to analyze the sentiment level in Shakespeare’s *Macbeth*. As *Macbeth* is a tragedy, as a reader, you would expect the negative sentiment level to increase and the positive level to decrease or stay the same closer to the play’s end. A scene would be a natural unit of analysis, but scenes are too long (on the VADER scale); single sentences, on the other hand, are usually the right size. Also, they can be easily extracted with a standard sentence tokenizer or through the corpus reader’s sentence tokenizing interface (function ``sents()``, see [Subsection 8.1.3](#) on p. 149).

Combine all words in each sentence into one string with ``str.join()`` and feed the string into the analyzer in a loop. Disregard short sentences (two or fewer words) because they most likely refer to scene numbers.

You can smoothen the scores before plotting them with the Pandas rolling filter ``DataFrame.rolling(n).func``. The filter takes each ``n`` consecutive rows in a dataframe and applies the function ``func`` to them. In our case, the function is ``.mean`` and $n = 25$.

```

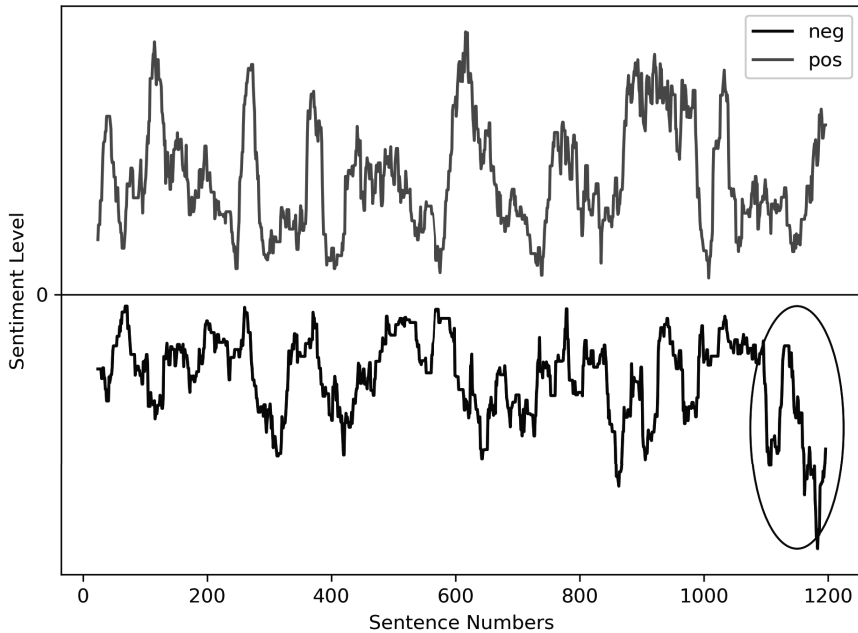
7  from nltk.sentiment.vader import SentimentIntensityAnalyzer
8
9  text = nltk.corpus.gutenberg.sents("shakespeare-macbeth.txt")
10 vader = SentimentIntensityAnalyzer()
11 scores = pd.DataFrame(vader.polarity_scores(" ".join(s))
12                        for s in text if len(s) > 2)
13 scores["neg"] = - scores["neg"]
14
15 smooth_scores = scores[["neg", "pos"]].rolling(25).mean()
16 ax = smooth_scores.plot()
```

[Figure 16.2](#) shows the traces of the negative and positive sentiment scores. Both scores behave somewhat randomly, but the negative score nosedives over the last one hundred sentences (circled on the chart), which is consistent with readers’ expectations: a tragedy should end on a negative note.

NLTK allows training specialized analyzers (e.g., ``SentimentAnalyzer`` from module ``nltk.sentiment``). While more flexible and fine-tuned to the research topic, these require the knowledge of advanced statistics and Naïve Bayesian classifiers and are outside of the book’s scope.

16.7 Topic Modeling

The Gensim library provides the Latent Dirichlet Allocation (LDA) algorithm for topic modeling. We will test it using the same Lewis Carroll’s *Alice’s Adventures in Wonderland*.

**FIGURE 16.2**

Positive and negative sentiment levels in *Macbeth*.

In the first stage, split the text of the tale into paragraph-size blocks to provide enough training data to LDA (it requires more than one document to operate) and preprocess them. The result is a list of tokenized, PoS-tagged paragraphs converted to lowercase and cleansed of the stopwords and punctuation.

In the Listing on p. 313:38, ``id2word`` is a ``gensim.corpora.Dictionary`` object, essentially a dictionary, that implements mappings between normalized, preprocessed words and their numerical identifiers. It takes a list of words or lemmas and enumerates and stores them. It can also be used to look up words by their IDs. In the example, ``id2word`` has 1,820 ID/word pairs, the number of unique lemmas in the text.

The function ``id2word.doc2bow(text)`` repeated in the list comprehension on line 39 converts a text block into a bag-of-words. A bag-of-words is a list of two-element tuples in which the first element is the index of a word in the ``Dictionary``, and the second is the number of occurrences of the word in the text. Thus, the bag-of-words ``[(8, 1), (25, 1), (29, 3), ..., (69, 1)]`` represents a text with one occurrence of the words “nothing,” “mind,” and “wonder” each, and three occurrences of the word “rabbit.”

The LDA algorithm, as such, is implemented in the function ``LdaModel()`` on line 42. The function takes the list of bags-of-words, the ``Dictionary``, and the number of topics to discover (defaults to 100). Choosing the right number of topics is crucial and often requires experimentation or domain knowledge to ensure the topics are meaningful and interpretable.

LDA is a generative probabilistic model that assumes that each document is a mixture of a few topics and that each word's presence is attributable to one of the document's topics. Since the model is probabilistic, it generates somewhat different results every time you run it.

Listing 16.2

lda.py

```

31 blocks = text.split("\n\n")
32 pre_blocks = [preprocess(block) for block in blocks]
33
34 # LDA itself
35 from gensim.corpora import Dictionary
36 from gensim.models import LdaModel
37
38 id2word = Dictionary(pre_blocks)
39 bows = [id2word.doc2bow(text) for text in pre_blocks]
40
41 # The number of topics (5) is somewhat arbitrary
42 lda_model = LdaModel(bows, id2word=id2word, num_topics=5)
43
44 # Visualization
45 pprint(lda_model.print_topics())
46 import pyLDAvis
47 vis = pyLDAvis.gensim.prepare(lda_model, bows, id2word)
48 pyLDAvis.save_html(vis, "LDA_Visualization.html")

```

Each topic is eventually output as a string of the form `'0.008*"head" + 0.008*"way" + 0.008*"time" + 0.008*"minute" + 0.008*"queen" + 0.007*"king" + 0.007*"right" + 0.006*"run" + 0.006*"eye" + 0.006*"dear"'`. This particular string defines a topic as a mix of the words “head,” “way,” “time,” “minute,” “queen,” “king,” “right,” “run,” “eye,” and “dear” in the respective proportions.

The results of the Alice experiment are presented and explained in [Table 8.3](#) on p. 162. You can also use the module `pyLDAvis` for advanced visualization in a Web browser.

16.8 What's Next

This chapter was the last. Your journey towards data science is over. We hope you enjoyed reading the book as much as we enjoyed writing it!

16.9 Programming Exercises

Exercise 26. What is the fraction of English male and female names (as defined in ``nltk.corpus.names``) that are also “proper English” words (as defined in ``nltk.corpus.words``)? Report two numbers. Use sets and set operations.

Exercise 27. Write a function ``sentiment_score(text, sent_dict)`` that calculates the sentiment score of the ``text`` using predefined word sentiments in the dictionary ``sent_dict``. Create a dictionary where keys are words and values are sentiment scores (e.g., ``{"happy": 1, "sad": -1}``). Ignore words not present in your sentiment dictionary and treat the sentence as neutral if its overall score is 0. The function shall sum the sentiment scores of words in the sentence and return -1, 0, or 1 based on the sign of the total sentiment score. Test your function.

Exercise 28. Construct and visualize a network of texts from the ``nltk.corpus.gutenberg`` corpus. Let each node ``N`` represent a text in the corpus. Let the weight of an edge between ``N1`` and ``N2`` be the Jaccard similarity⁷ of the sets of lemmatized and converted to the lowercase alphabetic non-stopwords in ``N1`` and ``N2``. Use ``nx.community.louvain_communities()`` to identify and label network communities.

⁷ $\text{Jaccard}(S_1, S_2) = \frac{\text{len}(S_1 \& S_2)}{\text{len}(S_1 \cup S_2)}$

Conclusion

As we draw the curtains on our exploration of data analysis in Python, it is imperative to reflect on the journey we have undertaken and the paths that lie ahead. Our book unfolds along two distinct tracks, each serving a unique purpose in equipping you with the knowledge and skills needed to navigate the complexities of data analysis in the social sciences. In the “executive track,” we probe the conceptual underpinnings of data analysis methodologies, guiding you through the data analysis life cycle from inception to interpretation. Chapters dedicated to understanding data structures, exploratory analysis, hypothesis testing, regression modeling, classification, network analysis, and text analysis lay the foundation upon which you can construct their analytical frameworks, fostering a deeper appreciation of the methodologies driving social science research.

Complementing the executive track is the “technical track,” which provides you with a solid grounding in Python programming essentials. By unraveling the intricacies of Python syntax, data manipulation, visualization, and algorithm implementation, we empower you to translate theoretical understanding into actionable insights, bridging the gap between theory and practice in data-driven inquiry.

The significance of our book lies not only in its role as a pioneering resource bridging the worlds of data science and social science but also in its unique structure, which grants you the flexibility to design your learning paths according to your needs and interests. By focusing on essential Python programming concepts and commonly used analytical methods in social science research, we seek to empower a new generation of social science practitioners with the tools and techniques needed to navigate the data-rich landscape of the 21st century.

However, we must acknowledge the limitations inherent in our endeavor. Our book does not aim to provide comprehensive coverage of the Python language, nor does it explore every analytical method conceivable. We also refrain from delving deeply into the mathematical or statistical foundation of the methods discussed. Additionally, our focus on commonly used methods in social science research means we do not explore popular data science methodologies such as neural networks.

Furthermore, we recognize the importance of research design in social science inquiry—an aspect often overlooked in data analysis textbooks. While our book primarily focuses on the data analysis life cycle post-data collection, we acknowledge the critical role of robust research design in shaping the

validity and reliability of research findings. Topics such as sampling strategies, survey methodology, measurement techniques, and dealing with biases warrant deeper exploration and careful consideration in the context of social science research.

To augment your knowledge of Python programming or explore alternative analytical methods not covered in this book, you are encouraged to explore additional resources. However, it is crucial to remember that data analysis, despite its power, may not always be infallible and is susceptible to various caveats and limitations. Substantive theories, contextual knowledge, and domain expertise are indispensable to data analysis, guiding researchers in their quest for meaningful insights and actionable conclusions.

As we navigate the ever-expanding landscape of data-driven inquiry, may we heed the call for talents who can seamlessly blend domain expertise with data analysis prowess. In bridging the gap between theory and practice, Python and social science, may we embark on a journey of discovery, innovation, and positive change.

Thank you for accompanying us on this odyssey through the intersection of data science and social science. May your analyses be insightful, your insights profound, and your impact enduring.

Happy analyzing!

Discussion Questions

1. How do you anticipate the knowledge acquired from this book will impact your approach to social science research?
2. How might a social scientist supplement their knowledge to address complex research questions that go beyond the scope of this book?

A

Solutions to Select Exercises

This chapter gives solutions to some of the programming exercised in the Technical Track, typically one problem per chapter.

Solution (Exercise 1 on p. 187). “Multiply” the number of spacer lines above and below the text by ``v_marg``. “Multiply” the number of space characters before and after the text by ``h_marg``. Note how a long line is “broken” into two lines with a backslash symbol ``\`` at the very end of line 12.

Listing A.1

better-frame.py

```
1 # Define style elements
2 h_marg = 2
3 v_marg = 2
4
5 # Define the text
6 text = "Boston"
7
8 # Construct the frame
9 width = len(text) + 2 * h_marg + 2
10 hline = "*" * width + "\n"
11 spacer = (" " * (width-2) + "*" + "\n") * v_marg
12 text = "*" + (" " * h_marg) + text\
13         + (" " * h_marg) + "*" + "\n"
14 frame = hline + spacer + text + spacer + hline
15
16 # Display the framed text
17 print(frame, end="")
```

Solution (Exercise 3 on p. 187). Count the number of occurrences of “o”, ``ohs``, and replace ``ohs-1`` of them.

Listing A.2

```
1 data = input("Enter a string: ")
2 ohs = data.count("o") # Number of occurrences
3 replaced = data.replace("o", "*", ohs - 1)
4 print(replaced)
```

Solution (Exercise 6 on p. 215). Create an uppercase version of a text file.

Listing A.3

```

1  # Request the file name from the user
2  filename = input("Enter the file name: ")
3
4  # Open the file in read mode
5  with open(filename, "r") as infile:
6      # Read the content
7      content = infile.read()
8
9  # Convert the content to upper case
10 upper_content = content.upper()
11
12 # Create a new filename
13 new_filename = "upper-" + filename
14
15 # Open the new file in write mode
16 with open(new_filename, "w") as outfile:
17     # Write the upper case content to the new file
18     outfile.write(upper_content)

```

Solution (Exercise 13 on p. 233). Report “Social support” outlier.

Listing A.4

```

1  THRESHOLD = 2
2  ss = happy21["Social support"]
3  std = ss.std()
4  mean = ss.mean()
5  condition = (ss - mean).abs() >= THRESHOLD * std
6  outliers = happy21.loc[condition, "Country name"].unique()
7  print(outliers)
8  #> ['Afghanistan' 'Benin' 'Ivory Coast' 'Lebanon'
9  #> 'Morocco']

```

Solution (Exercise 15 on p. 248). Identify normally distributed variables in 2021. First, obtain the subset of the dataframe that consists only of numerical variables. The column “year” is formally numerical but does not change in 2021 and should also be dropped. At this stage, you do not need the contents of the columns, only their names.

Listing A.5

```

1  num_columns = happy_full21.select_dtypes(include=np.number)\
2                      .drop(columns="year")

```

Next, organize a “for” loop that applies that Shapiro–Wilk test to each pre-chosen column after eliminating the missing values and displays the p-values.

Listing A.6

```

1  for col in num_columns:
2      result = sts.shapiro(nums[col].dropna())
3      print(f"{col:35}: {result.pvalue:.3f}")
4  #> Life Ladder : 0.051
5  #> Log GDP per capita : 0.008
6  #> Social support : 0.000
7  #> Healthy life expectancy at birth : 0.000
8  #> Freedom to make life choices : 0.000
9  #> Generosity : 0.025
10 #> Perceptions of corruption : 0.000
11 #> Positive affect : 0.000
12 #> Negative affect : 0.003
13 #> Confidence in national government : 0.056

```

The only normally distributed variables in 2021 are the “Life Ladder” and “Confidence in national government.”

Solution (Exercise 18 on p. 267). Identify the matching columns by checking if their data type is on the list and create a boolean array. Use the array to extract the column names.

Listing A.7

```

1  def continuos_vars(df):
2      matching_columns = df.dtypes.isin(("int", "float"))
3      return df.columns[matching_columns].tolist()

```

Solution (Exercise 21 on p. 281). Write a complete Python program that prints every odd-numbered line a large text file that does not start with a pound sign. The program must proceed line by line without loading the entire file into memory. The program must use generators.

Listing A.8

```

1  with open("large_text_file.txt") as infile:
2      for i, line in enumerate(file, start=1):
3          if i % 2 == 1 and not line.startswith("#"):
4              print(line, end="")

```

Solution (Exercise 24 on p. 297). Annotate the Zakhary’s Karate Club network nodes by assigning to them the attribute “max_clique”. The attribute’s value is 1 for the nodes belonging to the network’s largest clique and 0 otherwise. Draw the network and color the nodes based on the attribute’s value.

Listing A.9

```

1 cliques = nx.enumerate_all_cliques(karate)
2 largest_clique = sorted(cliques, key=len)[-1]
3 markers = [(1 if n in largest_clique else 0) for n in karate]
4 nx.set_node_attributes(karate, markers, "max_clique")
5 nx.draw_networkx(karate, node_color=markers)
6 plt.axis("off")

```

Solution (Exercise 27 on p. 314). Write a function `sentiment_score(text, sent_dict)` that calculates the sentiment score of the `text` using predefined word sentiments in the dictionary `sent_dict`. Create a dictionary where keys are words and values are sentiment scores (e.g., `{"happy": 1, "sad": -1}`). Ignore words not present in your sentiment dictionary and treat the sentence as neutral if its overall score is 0. The function shall sum the sentiment scores of words in the sentence and return -1, 0, or 1 based on the sign of the total sentiment score. Test your function.

Listing A.10

```

1 def sentiment_score(text, sent_dict):
2     # Tokenize the text into words
3     words = text.lower().split()
4     # Calculate the sum of sentiment scores
5     total_score = sum(sent_dict.get(word, 0) for word in words)
6     # Determine the sentiment based on the total score
7     return 1 if total_score > 0 else -1 if total_score < 0 else 0
8
9     # Testing the function with the predefined sentiment dictionary
10    sent_dict = {"happy": 1, "joy": 1, "pleased": 1, "sad": -1,
11                "unhappy": -1, "miserable": -1}
12
13    # Test sentences
14    test_sentences = [
15        "I am very happy and joyous.",
16        "This is a sad and miserable day.",
17        "I am neither happy nor sad, just neutral."
18    ]
19
20    # Test the function with the sentences
21    for sentence in test_sentences:
22        print(sentiment_score(sentence, sent_dict))

```

Bibliography

- [BA99] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.
- [BEF02] S.P. Borgatti, M.G. Everett, and L.C. Freeman. Ucinet for Windows: Software for social network analysis. Analytic Technologies, 2002.
- [BGLL08] V.B. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *Journal of Statistical Mechanics: Theory and Experiment*, (10):1000, 2008.
- [Bha23] S. Bhaimiya. An Asian MIT student asked AI to turn an image of her into a professional headshot. it made her white, with lighter skin and blue eyes, 2023.
- [BKL09] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O’Reilly Media, Inc., 2009.
- [DGG41] A. Davis, B.B. Gardner, and M.R. Gardner. *Deep South*. University of Chicago Press, Chicago, IL, 1941.
- [ER59] P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.
- [Gib13] D. M. Gibler. *International military alliances, 1648–2008*. CQ Press., 2013.
- [God] H. Godman. How long does the flu last? <https://www.health.harvard.edu/staying-healthy/how-long-does-the-flu-last>. Accessed on May 22, 2023.
- [HLS⁺22] J.F. Helliwell, R. Layard, J.D. Sachs, J.-E. De Neve, L.B. Akinin, and S. Wang. World Happiness Report 2022, 2022.
- [HSS08] A. Hadberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, August 2008.

- [kag21] Machine learning and data science survey. Kaggle, 2021. <https://www.kaggle.com/competitions/kaggle-survey-2021/overview>.
- [Ker14] D. Kernler. A visual representation of the empirical (68-95-99.7) rule based on the normal distribution, 2014. https://commons.wikimedia.org/wiki/File:Empirical_Rule.PNG.
- [KK05] Adam Kuper and Jessica Kuper, editors. *The Social Science Encyclopedia*. Routledge, New York, NY, 2005.
- [LH14] J. Leskovec and E. Horvitz. Geospatial structure of a planetary-scale social network. *IEEE Transactions on Computational Social Systems*, 1(3):156–163, 2014.
- [Lla18] E. Llaudet. The mobilizing and demobilizing effects of political TV ads: A midterm election study. *Electoral Studies*, 54:226–236, 2018.
- [MB16] A. Mrvar and V. Batagelj. Analysis and visualization of large networks with program package Pajek. *Complex Adaptive Systems Modeling*, 4:1–8, 2016.
- [New06] M.E.J. Newman. Modularity and Community Structure in Networks. *PNAS*, 103(23):8577–8696, 2006.
- [Sha11] R Shah. Future Tech Jobs: We Need Social Science Graduates. *Forbes*, 2011. <https://www.forbes.com/sites/rawnshah/-2011/06/22/future-tech-jobs-we-need-social-science-graduates/?sh=4b9b29353f26>.
- [Vig] T. Vigen. Spurious correlations. <https://tylervigen.com/spurious-correlations>. Accessed on March 30, 2023.
- [Wal23] J. Walrack. Where do I fall in the American economic class system? US News, 2023. <https://money.usnews.com/money/personal-finance/family-finance/articles/where-do-i-fall-in-the-american-economic-class-system>.
- [WS98] D. J. Watts and S. H. Strogatz. Collective Dynamics of “Small-World” Networks. *Nature*, 393(6684):440–442, 1998.
- [Zac77] W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 452–473, 1977.
- [Zin18] D. Zinoviev. *Complex Network Analysis in Python*. Pragmatic Bookshelf, 2018.

General Index

- 68–95–99.7 rule, [62](#)
- absolute frequency, [34](#)
- accessor, [201](#), [206](#), [225](#), [287](#), [309](#)
- accumulator, [237](#)
- accuracy, [119](#), [122](#), [278](#)
- actor, [127](#)
- adjacency matrix, [288](#)
- adjacency matrix, [132](#), [133](#)
- Albert, Réka, [285](#)
- algorithm, [3](#), [252](#)
 - lazy, [276](#)
- Anaconda, [170](#), [186](#)
 - Navigator, [186](#)
- ANOVA, [67](#), [239](#), [242](#)
 - one-way, [67](#), [242](#)
 - two-way, [67](#), [68](#), [243](#)
- API, *see* [application programming interface](#)
- application programming interface,
 - [14](#), [20](#)
- arc, [127](#)
- argument, [172](#), [249](#)
- array, [207](#)
- association, [49](#)
 - spurious, [49](#)
- attention mechanism, [158](#)
- attribute, [201](#), [302](#)
- AUC, [123](#), [280](#)
- augmented operator, [306](#)
- bag-of-words, [312](#)
- Barabási, Albert-László, [285](#)
- bar chart, [36](#)
- Basic English, [303](#)
- BeautifulSoup, [302](#)
- Berners-Lee, Tim, [301](#)
- betweenness centrality, *see* [centrality](#)
- bigram, [155](#)
- bin, [204](#)
- bond, [127](#)
- Bool, George, [177](#)
- Boolean, [29](#)
- boolean indexing, [200](#), [202](#), [203](#),
 - [207](#)
- bootstrapping, [70](#), [245](#)
- bootstrap samples, [72](#)
- break, [236](#)
- bridge, [127](#)
- Brown Corpus, [149](#)
- browser, [302](#), [313](#)
- Broyden–Fletcher–Goldfarb–Shanno algorithm, [275](#)
- Carroll, Lewis, [161](#), [162](#), [299](#), [302](#),
 - [311](#)
- categorization, [161](#), [204](#)
- category, [204](#), [268](#)
- causality, [52](#)
- causal analysis, [16](#)
- centrality
 - betweenness, [136](#), [294](#)
 - closeness, [135](#), [294](#)
 - degree, [136](#), [294](#)
 - eigenvector, [137](#), [294](#)
- central limit theorem, [71](#)
- central tendency, [41](#), [45](#)
- character
 - special, [174](#)
- chi-squared, [69](#)
 - contribution, [70](#)

- chi-squared test, 244, 245
- chi-squared test, 68, 106
- class, 268
- classification, 101, 268
- classification report, 279
- classifier, 116
 - Naïve Bayesian, 160, 268, 311
- clique, 138, 292
 - maximal, 138, 139, 292
 - maximum, 138
- closeness centrality, *see* centrality
- clustering coefficient, 294, 295
- clustering coefficient, 137
- CNA, *see* Complex Network Analysis
- co-occurrence, 161
- COCA, 149
- coefficient, 75
- comment, 172
- community detection, 138, 140, 141
- complete separation, 106
- complex network, 128
- Complex Network Analysis, 128, 132
- compressed sparse row, 288
- concatenate, 211
- conditional statement, 217
- confounder, 55
- confusion matrix, 116, 117, 279
- connected component, 131
- connection, 127
- constructor, 193
- container, 181, 189, 192, 203, 218, 236
- content curation, 156
- context, 158
- context manager, 194
- contingency table, 245
- contingency table, 69
- convergence, 103
- core, 139, 140
- core-periphery analysis, 138, 139, 292
- corona, 139
- corpus, 147, 149, 161, 305, 311
- Correlates of War dataset, 286
- correlation, 12, 16, 48, 87, 162, 229
- counterterrorism, 137
- cross-tabulation, 245
- cross-validation, 122
- crust, 140
- CSV, 21, 133, 134, 194, 195, 197
- cumulative odds, 111
- cutpoint, 111
- cycle, 131
- data
 - long, 24
 - wide, 24
- data alignment, 206
- data filtering, 207
- data imputation, 209
- dataframe, 23, 34, 182, 189, 197
- data type, 25, 28
 - bool, 182
 - dict, 182
 - float, 182
 - int, 182
 - list, 182
 - ndarray, 182
 - NonType, 182
 - set, 182
 - str, 182
 - tuple, 182
- Davis, Allison, 142
- decision threshold, 123
- decision tree, 268
- Declaration of Independence, 155
- deduplication, 301
- deep learning, 158
- degree, 130
 - weighted, 131
- degree centrality, *see* centrality
- degree of freedom, 65, 67, 69, 80
- density, 137
 - global, 137, 294
 - local, 294
- descriptive analysis, 15, 34
- dictionary, 189

- dictribution
 - normal, 63
- digraph, 284
- directed network, *see* network
- distance metric, 114
- distribution, 42
 - chi-squared, 60, 106
 - chi-squared distribution, 69
 - F-distribution, 60, 67
 - multimodal, 44
 - normal, 60, 62, 65, 67
 - skewed, 42
 - symmetrical, 42
 - t-distribution, 60, 65
 - unimodal, 44
- docstring, 255
- dummy, 91, 103
- dyad, 139, 292
- edge, 127, 129
 - attributes, 129
 - weight, 130, 143
- edge list, 132–134
- eigenvector centrality, *see* centrality
- element, 127
- entity, 148
- epidemiology, 137
- Erdős, Paul, 285
- error, 81
- error message, 171, 219, 220
- ETL, *see* extract-transform-load
- Euclidean distance, 115, 277
- Euripides, 137
- Excel, 169, 197
- exception, 220, 236
 - handle, 220
 - raise, 220
- explanatory power, 10
- exploratory analysis, 16, 34, 47
- expression, 238
- extract-transform-load, 14
- F1 score, 120
- factor analysis, 295
- fall-out, 120, 123, 280
- false negative, 117
- false positive, 117, 119
 - rate, 120
- filter(), 304
- filtering, 207
- fintech, 5
- Flickr 8k Dataset, 150
- float, 29
- fold, 121, 278
- for, 236
- formatted string, 184
- frequency analysis, 155
- frequency distribution, 34
- function, 172
 - polymorphic, 251
 - user-defined, 249
- Gardner, Burleigh, 142
- Gardner, Mary, 142
- GCC, *see* giant connected component
- generalizability, 54, 55
- generalized linear models, 101
- generator, 268, 269, 282, 290, 294
- generator expression, 271
- Gensim, 311
- Gephi, 297
- GEXF, 134, 286
- giant connected component, 132, 289, 290
- GML, 134, 286
- goodness-of-fit, 80, 105, 244
- googol, 185
- GPT, 157, 158
- graph, 283
- GraphML, 134, 286
- Graphviz, 297
- graph theory, 127, 140
- Gutenberg Corpus, 161
- harmonic mean, 120
- heteroscedasticity, 84, 243
- heuristics, 150
- histogram, 38, 224, 240

- homoscedasticity, 83, 242
- HTML, *see* [HyperText Markup Language](#)
- Hugging Face, 157
- Hugo, Victor, 285
- human coder, 159
- hyperparameter, 278
- HyperText Markup Language, 148, 163, 301
- hypothesis, 10, 56
 - alternative, 56
 - null, 56
- hypothesis testing, 13, 56
 - left-tailed, 59, 63
 - one-tailed, 59
 - right-tailed, 59
 - two-tailed, 59, 63
- IDE, *see* [integrated development environment](#)
- imputation, 30
- in, 236
- indegree, 130
- index, 199
- inferential analysis, 16
- information retrieval, 161
- integer, 29
- integrated development environment, 169, 170, 224
- interaction, 92
 - two-way, 99
- interaction effect, 93
- intercept, 77
- interface, 249
- interquartile, 86
- isolate, 131, 132, 135, 284
- iteration, 234
- iterator, 269
- Jaccard similarity, 314
- join
 - inner, 212
 - outer, 212
- JSON, 194, 197
- k-nearest neighbors, 113, 268, 276, 277, 280
- Kaggle, 20
- key, 192
- keyword, 161
- KNeighborsClassifier(), 277, 278
- label, 129
- Lancaster University, 153
- language detection, 147, 149
- Latent Dirichlet Allocation, 311
- Latent Dirichlet Allocation, 161, 313
- latent theme, 161
- layout, 295
 - circular, 295
 - Fruchterman–Reingold, 295
 - Kamada–Kawai, 295
 - spiral, 295
 - spring, 295
- lazy, 268, 270, 282
- LDA, *see* [Latent Dirichlet Allocation](#)
- lemma, 153, 307
- lemmatization, 152, 153, 303, 306
- lemmatizer, 153, 307
 - WordNet, 307
- Les Misérables, 285
- levels of measurement, 204
 - interval, 27
- level of measurement, 25
 - categorical, 27
 - continuous, 27
 - nominal, 25
 - ordinal, 25
- likelihood ratio test, 105, 113
- linearity, 106
- link, 127
- link function, 101
- list, 270
 - parallel, 270, 274
- list comprehension, 238
- listing, 180
- list comprehension, 271
- literature review, 9

- LiveJournal, 160
- LLR test, *see* [log-likelihood ratio test](#)
- log-likelihood ratio test, 105
- logistic regression, 268, 272
 - multinomial, 107
 - ordered, 111, 112, 275
- log odds, 102, 104
- log transformation, 31
- loop, 234, 268
 - for, 234, 236, 246
 - while, 234, 246
- Louvain algorithm, 141, 294
 - resolution, 142
- LR test, *see* [likelihood ratio test](#)
- Macbeth, 311
- machine learning, 116, 158, 161
- macro-average, 122
- main effect, 93
- Manhattan distance, 277
- Mann, Henry, 246
- Mann–Whitney U test, 245
- Maximum Likelihood Estimation, 103
- mean, 30, 41, 42, 60, 226, 228
- median, 30, 41, 42, 228
- membership, 237
- Minkowski metric, 277
- missing value, 206, 209
- missing value, 29
- MLE, *see* [Maximum Likelihood Estimation](#)
- mode, 30, 41, 227
- modularity, 140, 141, 293
- module, 185
 - calendar, 185
 - collections, 185, 222
 - csv, 185
 - datetime, 185
 - itertools, 185
 - json, 185
 - math, 185, 186
 - matplotlib, 197, 224
 - networkx, 283
 - numpy, 182, 186, 197, 200, 207
 - pandas, 182, 186, 197, 222, 223
 - pickle, 185, 196
 - pyLDAvis, 313
 - random, 185, 186
 - re, 185
 - scipy, 197, 210, 239
 - stats, 239, 241
 - sklearn, 197
 - SpaCy, 309
 - statsmodels, 239, 241
 - xml, 185
- monad, 292
- motif census, 292
- multicollinearity, 87, 103, 106, 295
- multidigraph, 284
- multigraph, 284
- MultiIndex, 214
- multinomial logistic regression, 273
- n-clique, *see* [clique](#)
- n-gram, 149, 155
- Naïve Bayesian, 160, 268, 311
- named entity, 158, 310
- named entity recognition, 158, 159, 303, 309
- Natural Language Processing, 147, 298, 300, 303
- Natural Language Toolkit, 150, 152, 161, 192, 303, 305, 306, 309, 310
- neighborhood, 138, 294
- NER, *see* [named entity recognition](#)
- network, 283
 - bipartite, 142, 143, 291
 - projection, 143, 291
 - community, 140, 293
 - directed, 130
 - fully connected, 135, 137, 138, 292
 - hub-and-spoke, 136
 - tripartite, 291
 - undirected, 130
 - weighted, 130
- NetworkX, 283, 284, 287

- networkx
 - subgraph(), 290
- neural network, 268
- news aggregation, 156
- Newton–Raphson algorithm, 275
- NLP, *see* [Natural Language Processing](#)
- NLTK, *see* [Natural Language Toolkit](#)
- node, 127, 129
 - attributes, 129
 - endogeneous, 130
 - exogeneous, 130, 138
- nominal, 41, 90
- non-parametric approach, 59
- normalization, 184
- normalize, 116, 122
- normal distribution, *see* [distribution](#)
- NoSQL, 197
- null hypothesis, 239
- null hypothesis, 75, 113
- null parameter, 57, 61
- numpy, 247, 257, 288
- observation, 268
- observational method, 54
- odds, 102
- odds ratio, 105
- OfficeLibre, 197
- Ogden, Charles Kay, 303
- OLS, *see* [Ordinary Least Squares](#)
- ols(), 244
- onion decomposition, 139
- OpenAI, 157
- operator
 - index, 175
 - logical, 181, 217
 - not in, 265
 - relational, 181, 217
 - slicing, 176
 - unpacking, 243
 - vectorized, 207
- ordinal, 41, 90
- Ordinary Least Squares, 244
- Ordinary Least Squares, 257
- Ordinary Least Squares, 76, 77, 80, 82, 83, 87, 101, 257, 258, 265, 267
- outdegree, 130
- outlier, 15, 30, 47, 86, 103, 190, 229
- overfitting, 121, 279
- p-value, 61, 66, 67, 72, 76, 80, 82, 106, 239, 240, 244, 245
- package, 185
- PageRank, 309
- Paice, Chris, 153
- Pajek, 134, 286
- Pandas, 286
- pandas, 182
- parallel lists, *see* [lists](#)
- parallel lists, *see* [lists](#)
- parallel regression assumption, 112
- parallel slope assumption, 94
- parametric approach, 59
- parsimony, 113
- part-of-speech, 147, 152, 304, 307
 - tagging, 152, 303
- partition, 141, 293
- path, 131
 - geodesic, 131, 136
 - shortest, 131
 - simple, 131
- pathway, 127
- Patsy, 257, 258, 260, 262, 263, 272, 273, 275
- PennTreebank, 149
- periphery, 139
- pickling, 196
- pie chart, 40
- pip, 186
- plain text, 148
- point, 127
- polarity score, 311
- population, 12
- Porter, Martin, 153
- PoS tagging, *see* [part-of-speech tagging](#)
- precedence, 182
- precision, 120, 122, 279

- preferential attachment, 285
- preprocessing, 24
- principal component analysis, 295
- probability distribution, 101
- Project Gutenberg, 161
- proxy, 12
- pseudo-R-squared, 106
- puzzle, 9
- Python standard library, 185

- quantile, 228

- R-squared, 76, 80, 106
- Rényi, Alfréd, 285
- randomized controlled experiment, 53
- random forest, 268
- random sampling, 13
- range, 45
- recall, 120, 122, 123, 280
- Receiver Operating Characteristic curve, 123, 280
- regression, 75, 81
 - linear, 75, 244, 256
 - logistic, 102
 - logit, 102
 - multiple, 75, 87
 - multivariate, 75, 87
 - non-linear, 75
 - ordinary least squares, 272
 - simple, 75
- relative frequency, 35
- research question, 9
 - causal, 9
 - directional, 59
 - exploratory, 9
 - non-directional, 58
- residual, 79, 81, 83, 260
- Richards, Ivor, 303
- rich get richer, 285
- robust test, 16
- ROC curve, *see* Receiver Operating Characteristic curve
- rolling filter, 311
- root, 306

- rule, 159
- R programming language, 257

- sample size
 - large-N, 13
 - small-N, 13
- sample statistic, 61
- sampling distribution, 71
- scaler, 277
- scatter plot, 48
- SciKit-Learn, 257, 276
- selection bias, 55
- self-loop, 129, 133
- sensitivity, 120
- sentence, 150
- sentence tokenization, 150, 311
- sentence tokenizer, 150
- sentiment, 161
- sentiment analysis, 155, 160, 303, 310
- sentiment analyzer, 160
- separator, 191
- series, 182, 189, 197
- set, 291, 298, 299, 301, 303
 - difference, 299
 - empty, 298
 - intersection, 299
 - union, 299
- Shakespeare, William, 311
- Shapiro–Wilk test, 85, 239, 240
- Shapiro–Wilk test, 64–66, 278
- sigmoid, 103, 272
- site, 127
- six degrees of separation, 285
- slice, 176
- slope, 77, 80, 82
- small world, 285
- smart indexing, 200
- SNA, *see* Social Network Analysis
- Social Network Analysis, 128
- sociogram, 127
- Socrates, 137
- Southern Women dataset, 142, 291
- spaCy, 159, 309
- specificity, 121

- spoke, 136
- spreadsheet, 169
- Spyder, 170
- SQL, 197
- standard deviation, 226, 228
- standardize, 84
- standard deviation, 46, 60–62
- standard error, 61, 80, 82
- statement
 - body, 218
- station, 127
- statistical significance, 80
- stemmer, 153, 306
 - Lancaster, 153, 306
 - Porter, 153, 306
- stemming, 152, 153, 303, 306
- stereotyping, 124
- stop word, 149, 153, 161, 190, 300
- string, 28
 - empty, 180, 198
- Strogatz, Steven, 285
- strongly connected component, 132
- subclassification, 56
- subnetwork, 138
- summarization, 155, 156
 - abstractive, 157, 162, 309
 - extractive, 156, 157, 303, 307
- summary, 156–158
- supervised machine learning, 159
- support, 279
- survival bias, 55
- synset, 149
- syntactic sugar, 306

- t-statistic, 66, 80
- t-test, 65, 241
 - independent samples, 66, 241
 - paired, 66, 241
- tag, 148, 302, 304
- TensorFlow, 157
- testing dataset, 116
- testing set, 278
- test statistic, 60–63
- tetrad, 139
- text preprocessing, 303

- TextRank, 309
- text mining, 147
- text processing pipeline, 147, 153
- tie, 127
- tokenization, 150
- topic, 161
 - discovery, 161, 162
 - modeler, 161
 - modeling, 161, 311
- TPP, *see* text processing pipeline
- training dataset, 116
- training set, 278
- transformer, 157, 158
- translation, 194
- treatment group, 55
- trend, 161
- triad, 139, 292
- trigram, 155
- true negative, 117
 - rate, 121
- true positive, 117
 - rate, 120
- truth table, 181
- tuple, 190
 - named, 239
- Twitter, 14, 20, 160, 310

- UCINET, 134, 286
- undirected network, *see* network
- unit, 127
- unit of analysis, 13
- unpacking, 243

- VADER, 160, 310
- valence, 160, 310
- validation, 121, 278
 - cross-validation, 121, 278
 - hold-out, 121, 278
- variability, 45
- variable, 11
 - control, 11
 - dependent, 11
 - global, 254
 - independent, 11
 - intervening, 11
 - local, 253

- variance, [45](#), [67](#), [228](#)
- vector, [207](#)
- Venn diagram, [299](#), [300](#)
- vertex, [127](#)
- visualization, [295](#)

- Watts-Strogatz graph, [285](#)
- weakly connected component, [132](#)
- web scraping, [14](#), [21](#)
- weighted network, *see* [network](#)
- word, [151](#), [155](#)
 - slang, [160](#)

- WordNet, [149](#), [307](#)
- word tokenization, [148](#), [150–152](#)
- word tokenizer, [151](#), [155](#)

- XML, [194](#), [197](#), [286](#)
- X (social media), [310](#)

- Z-score, [64](#), [241](#)
- Z-test, [63](#), [64](#), [72](#), [240](#), [241](#)
- Zachary’s karate dataset, [127](#), [282](#),
[284](#), [285](#), [290](#), [292](#), [297](#)
- Zachary, Wayne, [127](#)
- zettabyte, [298](#)